Queen Mary University of London

School of Electronic Engineering and Computer Science

# Latency Based Approach for Characterization of Cloud Application Performance

Hamed Saljooghinejad

Submitted in part fulfilment of the requirements for the degree of
Master of Philosophy
October 2015

To my wife **Roshanak**

who gives depth to my life

# Acknowledgements

I would like to thank my supervisors Dr Felix Cuadrado and Professor Steve Uhlig for their wise guidance and supervision and for being so patient with me.

My sincere gratitude to Dr Gareth Tyson who always supported us with his wise comments and interesting discussion.

I would like also to thank all the friends in our research group specially Marjan, Sabri, Amna, Kishan and Shan.

Finally, I thank my wife and my parents who have always supported me even in the difficult moments of my life.

**Hamed Saljooghinejad**

**October 2015**

# Publication:

- Hamed Saljooghinejad, Felix Cuadrado and Steve Uhlig. *Let Latency Guide You: Towards Characterization of Cloud Application Performance*. IEEE 7th International Conference on Cloud Computing Technology and Science (**IEEE CloudCom 2015**).

# Abstract

Public cloud infrastructures provide flexible hosting for web application providers, but the rented virtual machines (VMs) often offer unpredictable performance to the deployed applications. Understanding cloud performance is challenging for application providers, as clouds provide limited information that would help them have expectations about their application performance. In this thesis I present a technique to measure the performance of cloud applications, based on observations of the application latency. I treat the cloud application as a black box, making no assumption about the underlying platform. From my measurements, I can observe the varying performance provided by the different VM profiles across well-known commercial cloud platforms. I also identify a trade-off between the responsiveness and the load of the measured servers, which can help application providers in their deployment and provisioning.

# Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

Cloud computing offers on-demand computing available all around the world. Many start-ups and medium size application providers who do not own any infrastructure tend to deploy their applications on public cloud infrastructures because of on-demand resources, ease of deployment and instant scalability of these platforms [VRMCL08].

Choosing public cloud to host the applications introduces new issues as these platforms function in ways that are significantly different from traditional on-premise servers. In such platforms, Cloud Service-Level Agreements (SLAs) provide guarantees only on infrastructure reliability. They contain no performance-related clauses [Ama14a], and Virtual Machine instance types hide underlying hardware details. Consequently, providers don't know the expected performance of their application as the result of virtualization impact on individual components when deployed on the cloud infrastructure [WN10, JRM+10].

While this is challenging, one of the major concern of application providers is meeting the stringent low response times for interactive applications (e.g., customer facing web applications). Response time is proven to have a direct effect on business revenue. Amazon found that every 100ms of latency costs 1% in sales [Lin06], and eBay and Google have reported similar findings about the impact of user latency on revenue [Dix09, Ham09, SB09]. Therefore, minimizing the latency and preserving the delay of serving the users requests under an acceptable latency value is challenging and important.

Cloud providers offer different types of servers (VMs) across various geographical locations in world. Having a geo-distributed service provides potential opportunity to deploy applications on different locations across the world to benefit from low path latency. Therefore, geo-distributed applications need an effective way to direct client requests to a particular application replica in order to minimize user perceived latency.

User-perceived latency consists of two parts at minimum, the network delay to the application replica a user is connected to; and processing time of requests at application replica. To guarantee user perceived latency to be within a defined value, any solution must be able to estimate the above two delays. While estimating the network latency between users and replicas has been well researched in past, my focus is on *application processing time*. I aim to propose techniques and algorithms to estimate the corresponding delay and open the space for future research to guarantee and control such a delay for a target application.

The research question behind my work is: **Is it possible to estimate the load of a server by observing its response latency from a remote network location?**

To be able to answer the above question, first it is necessary to know how many requests my application can handle and what the corresponding delay will be at different loads. Understanding the performance of an application under load and its effects on application processing delay is challenging due to the underlying infrastructure that the application is deployed on, e.g., a more powerful node with more resources (e.g., CPU, Memory and I/O) handles more requests without causing extra delay and affecting the overall latency. Moreover, in a heterogeneous platform like cloud where cloud providers offer a variety of VM types that comprise various combinations of CPU, memory, storage and networking capacity, it is difficult to translate mix of resources to a value of performance that is meaningful at the application level.

To address the above challenge I propose an algorithm to understand the performance of the application server. My technique makes no assumption about the underlying infrastructure and internals of the application and explores the application performance in terms of throughput with respect to the corresponding response time.

The proposed technique will answer the following questions: 1) How many requests does the application handle without affecting the response time? (i.e., it estimates the maximum load a cloud application can sustain for a given VM while preserving defined latency SLA). 2) How much the response time will vary at different levels of load? (i.e., it identifies a trade-off between application performance and server responsiveness).

In conclusion, our contribution can help application provider to estimate the load of an application server from a remote network location. In fact using my proposed technique they can profile application performance by running our designed tool on different types of VMs on any cloud platforms. Such a profile includes the trade-off information between application performance and server responsiveness. This data can further be utilized to build a model for each profile which will be used further on to estimate the load at any time, i.e., having a model for a node, one can send a request at any time to the server from a remote node and measure the response time. The response time will then be fed to the model to estimate the server load.

***The rest of this thesis is organized as follows :***

In **Chapter 2**, I review some of the related works about performance evaluation of infrastructure and applications in the cloud as well as commercial and academic solutions that control and guarantee the server side performance of applications.

In **Chapter 3**, I propose a black-box methodology to measure cloud application performance. My methodology estimates the maximum load an application server can sustain at which the latency of the responses increases to an undesirable level. It samples the latency values at various loads and observes the trade-off between application performance and server responsiveness.

In **Chapter 4**, I implemented the methodology proposed in chapter 3, and utilized that to estimate the load a cloud application can sustain when it is deployed on different types of VMs in various cloud providers. Moreover, I identify trade-off between the throughput and latency of application servers. I also test different types of workloads and show throughput estimation in each case.

In **Chapter 5**, I propose possible avenues for future work based on the present results of this

thesis..

In **Chapter 6** I present the conclusion of this thesis.

# Chapter 2

# Related Work

In this chapter, I review some of the related works about performance estimation in cloud. I explain the methods and tools that have been proposed by researcher to benchmark the performance of the infrastructure and the application. Furthermore, I explain some of the other efforts to reduce the delay in serving the user requests. Finally, I illustrate the limitations of the current approaches and explain my contributions toward estimating the performance of cloud applications.

## 2.1 Performance Evaluation in the Cloud

Cloud computing offers different types of virtual machines (VMs) with varying combinations of CPU, memory, storage and networking capacity to choose the appropriate mix of resources for the applications. My work focuses on understanding the performance of applications on a given VM in the cloud. In this part I review different types of tools and studies related to performance evaluation of infrastructure and deployed applications. Different benchmarking tools exist at various levels of abstraction, from the low level system part like CPU, to the whole application, e.g., database system. Benchmarking and monitoring tools are used for evaluating and tracking the performance of both infrastructure and application. I review some of the benchmarking and monitoring tools and mention the difference between those tools and

my work.

### 2.1.1 Low level Benchmarking

With the advent of Cloud Computing, more and more providers offer Infrastructure-as-a-Service (IaaS) platforms. IaaS allows the customers to launch different types of virtual machines (VMs) with varying combinations of CPU, memory, storage and networking capacity to choose the appropriate mix of resources for the applications. To measure the performance of a VM often low level benchmarking tools (or micro-benchmarking) are used. Micro-benchmarks are mostly designed to stress each of the main computer resources individually (e.g., CPU, disk or network) provides one score that is used for comparison. Many open source and commercial tools are employed to evaluate the performance of the CPU, memory, storage devices and network components of a node (VM) in the cloud.

| **CPU** | **Memory** | **Disk** | **Network** |
|---|---|---|---|
| Ubench | Cachebench10 | Iozone | NetPerf |
| are Dhrystone | STREAM11 | bonniee++ | Ttcp |
| Whitestone | UnixBench | Dbench | NetSpec16 |
| Unixbench | GeekBench12 | | Iperf |
| SPEC CPU 2000 | Ubench | | |
| SPEC HPC 96 | | | |

Table 2.1: List of low level benchmarking tools

In Table 2.1, I have listed some of the micro-benchmarking tools that are used by researcher to benchmark each component of a node (VM) individually. I would show in the following sections, how these tools have been used by researchers to show the performance variations of instances in the existing public cloud platforms.

#### Usage of Low level benchmarking tools in cloud.

Low level benchmarking tools have been used in previous research studies for different purposes with regard to performance evaluation of VMs in the cloud:

**1)** *Comparison of offered VMs from different cloud providers:*

Li et al. [LYKZ10] compare multiple cloud providers using different types of micro-benchmarks. They ran different micro-benchmarks related to each individual resource like computation metrics (CPU, Memory), network metrics (network latency, bandwidth) and compared various results for various cloud platforms (e.g., Amazon AWS [Ama14b], Microsoft Azure [Azu14], Google AppEngine [GAE14], and Rackspace CloudServers [Rac14]). They claimed that there is no single winner on all metrics.

**2)** *Performance variation due to virtualization in cloud:*

Schad et al. [SDQR10] have leveraged different types of micro-benchmarks[Ube01, Bon01] and show high performance variation for most of their metrics related to CPU, disk I/O and network in Amazon EC2. Farley et al. [FJV+12] have used micro-benchmarks to show that heterogeneity in the underlying hardware and contention can cause different performance across even equivalent instances. They have also explored a heterogeneity-aware placement strategy that seeks out better performing instances. Wang et al. [WN10] have presented a measurement study on the impact of virtualization on the Amazon EC2 platform. They show that the network performance of nodes in EC2 is much more variable than that of non-virtualized clusters due to virtualization and processor sharing. Ou et al [OZN+] considered hardware heterogeneity within EC2, and within a single instance type and availability zone. Using micro-benchmarks in Table 2.1 and verifying hardware for each instance, the variation in performance across different instances with different hardware for CPU-intensive workloads can be as high as 60%. Barker et al. [BS10] analyzed the impact of virtualization on the performance of latency sensitive applications. They have quantified the jitter of CPU, disk, and network performance in regard to latency-sensitive applications in Amazon EC2 cloud. Studies like [OIY+10, IOY+11, JRM+10] use micro-benchmarking tools like [spe, Bon01, dbe, uni] to compare the actual performance difference between cloud computing and traditional high performance clusters so as to evaluate the applicability of cloud computing to scientific applications.

Overall, low-level benchmarking tools are useful to compare virtual and physical execution environments. However, such a tools can not measure the performance at the applications. This aspect complicates the use of low-level benchmarks for guiding deployment decisions of

applications.

### 2.1.2 Application level benchmarking

There is a wide range of application-level benchmarks, with each tool focussing on a specific family of applications. GridMix [gri], HiBench [HHD+10], and Berkeley WL Suite [CGGK11] are benchmarks designed for evaluating the Hadoop framework. MineBench [NOZ+06] benchmarks data mining algorithms. CloudRank-D [LZJ+12] offers a benchmark suite for various machine learning and data mining algorithms. CloudSuite [FAK+12] has collected individual benchmarking tools for different types of applications e.g. data analytics, and web search. The above benchmarking tools target batch-style applications, rather than interactive systems handling user requests. My focus is on user requests and dealing with corresponding response times to those requests. Cloudstone [SSS+08] is a benchmarking tool that deploys one web 2.0 application composed of front-end and back-end. Cloudstone gives several application-specific performance metrics (such as how many concurrent users can be logged-in and supported for a fixed dollar amount) for that specific application. The specific nature of this benchmark makes its applicability similar to the system level benchmarks previously described.

On-line transaction processing (OLTP) benchmarks are used to evaluate the performance of back-end databases. For example, TPC-C [tpc14] simulates a few types of transactions against a database and computes two types of results: pure performance (transactions per minute) and performance over price. Their main goal is to deliver a single performance number to compare different database systems and answer the question: which database system is the best for OLTP?. TPC-C is considered the reference benchmark for transactional databases. More recently, YCSB [CST+10] aims to fill a similar gap for NoSQL data store systems. These types of benchmarks are useful for performance comparison between multiple similar products (such as SQL-compliant databases). Their applicability is still limited to a family of similar applications.

In contrast, my technique computes an application-level metric (maximum number of requests per second while fulfilling a SLA) in an application agnostic way (as is demonstrated in the

evaluation section).

### 2.1.3 Monitoring tools

Performance monitoring tools usually provide monitoring and analysis of specific parameters of system and notification about critical changes in their status. Nagios [Nag14] monitors system metrics, network protocols and services. Processor load, disk usage, system logs, interactions and connectivity can also be monitored. Zabbix [Zab14] is another monitoring tool that tracks the status of CPU, memory, network, disk I/O, disk space and log files.

The purpose of monitoring tools is to monitor and alert users about the changes in server performance elements. Moreover, monitoring tools are by nature white-box, i.e., they have to access and track the status of internal elements of the system. The results of monitoring tools are difficult to translate into global application level-performance. However, the focus of my work is to provide a fine grain behaviour of performance at the application level. My approach is black-box, i.e., I treat the whole application server as a black-box with no assumption about internal of application and I employ the external values of latency from a remote node and explore the performance space of applications.

## 2.2 Controlling and Minimizing the Response Time

My research is motivated by the use of geo-distributed applications that are capable of being deployed at different geo-distributed cloud data centers. Keeping the response time under a desirable value in such scenarios is crucial for application providers. In the following section I review some of the related work about server selection which is employed to decrease the user response time when many replicas can serve the user requests around the world.

### 2.2.1 Server Selection

The simplest approach to distribute client requests among servers is the use of a round-robin algorithm [Bri95, CYC98]. But in the case of widely distributed servers across the globe this approach will end-up with high network latency and unsatisfactory delay for users' requests. Previous research focused on minimizing network path latency to user by selecting a server (among many replicas containing static content) that is closer to the user. The most well-known approach is DNS-based server selection [SCKB06, DMP+02]. It requires running a DNS server which tracks where the servers are running and measuring the network latencies and aims at selecting the replica which has minimum delay to the user. This is the method widely employed by CDN providers like [Aka14]. The primary objective of CDNs, as well as of mirroring and caching strategies, is to reduce the network latencies between the clients and the server they are accessing. This is done in two ways: Firstly, by directing clients to the nearest server [GS95, FBZA, KLL+97, WPP02, KM02] and secondly, by placing the most popular content on replicas closer to hot-spots [CKK02, QPV01]. Studies like [PAS+04, STA01], have shown that the use of DNS-based redirection techniques may lead to very high delays e.g., wrong approximation of DNS servers and thus may not be suitable for applications which require quick response to failures. Moreover, DNS-based redirection mechanisms are agnostic of application structure and unaware of application performance.

Latency estimation methods have been introduced in the past, and they can be employed in the server selection problem to redirect clients to a server with the lowest estimated latency. (e.g., redirecting clients to the nearest data centers would require Google to maintain latency from virtually every Web client in the Internet to each of its data centers [BP98]). The research community offeres a variety of techniques such as: direct network measurements [WSS05, KMS+09, GSG02], virtual coordinates [DCKM04, NZ02], structural models of path performance [FJJ+01, MIP+06], some hybrid of these approaches [AL09, FRE06] or commercial IP geo-location database [Neu14, PS01] to estimate network proximity. For example, since early 2005, Wikipedia has been using PowerDNS, in combination with the geo-backends [Geo04] to handle all DNS traffic [Wik05]. By using the geo-backend, incoming clients

can be directed to the nearest Wikipedia server (based on their geographic locations).

Increasingly, cloud computing provides an attractive alternative where cloud providers offer servers across the world, while allowing customers to design and implement their own services. Today, such customers are left largely to handle the replica-selection process on their own, with a limited support from individual cloud providers [Rou14, WAz14] and third-party DNS hosting platforms [Dyn14]. For example, cloud providers offer services to redirect clients to a VM server in a data center with lowest network latency [Rou14, WAz14]. Amazon Route 53 [Rou14] offers different services for server redirection, e.g., Weighted Round Robin or Latency Based Routing. Latency Based Routing can automatically route end-users to a cloud data center which has the lowest network path latency.

While the above research and services purely focus on selecting a replica with the lowest latency to users, studies like [RKK04] show and exploit the fact that the processing delay of dynamic content on moderately to highly loaded servers can exceed network delays by an order of magnitude. Therefore server load should be considered as a metric to select a replica.

Wendell et al. introduced Donar [WJFR10], a distributed approach for server selection which consists of multiple mapping nodes that is able to handle a diverse mix of clients. The mapping nodes could be HTTP ingress proxies that route client requests from a given locale to the appropriate data centers, the model adopted by Google and Yahoo. Or, the mapping nodes could be authoritative DNS servers that resolve local queries for the names of Web sites, the model adopted by Akamai and most CDNs. Each mapping node has only a partial view of the global space of clients. Their contribution is a decentralized algorithm to work with mapping nodes and redirect clients based on two policies, *network latency* and *server load*. This has not been addressed by the previous heuristic-based solutions [CYD97, CGP00, CCY00]. While valuable they assumed that the load on a server is known to mapping nodes but in fact estimating the load and its effect on latency (application processing delay) is challenging due to the heterogeneity of infrastructure, complexity of application structure, workload and the policy for storing and retrieving data.

## 2.3 Summary

In this chapter I have reviewed the methods and tools that have been used by researcher to benchmark the infrastructure and application performance. While valuable none of the methods have concentrated on estimating the performance with respect to latency that the user will perceive. Therefore the ability to estimate any performance value while considering the latency is crucial for any further solution that is going to be offered to control the response time of the application based on the its performance.

# Chapter 3

# Black-Box Latency Based Throughput Estimation

It is important for application providers to decide about provisioning enough servers to achieve the desired throughput while preserving acceptable latency. In this chapter I propose a methodology that leverages the statistical properties of latency(e.g., median) to detect the behaviour of a cloud application server under load. My black-box methodology estimates the workload a cloud application can sustain for a given latency on a given server.

## 3.1 Methodology

The main goal of my methodology is to observe the trade-off between application performance (in terms of operations per second) and server responsiveness (as observed through request/response latency). I observe the *server response time* across various workloads by remotely measuring the end-to-end latency.

### 3.1.1 Approach

My approach relies on the measured end-to-end latency of individual request/response pairs. This end-to-end latency includes the network latency, as well as the client and server side processing latencies. I generate controlled amounts of requests/responses at a constant rate, while monitoring the corresponding end-to-end latency values. The rationale is the following: Application servers are typically designed to absorb a given workload of requests. Hence, when the workload is below a given threshold, I expect the server latency to be low and relatively constant over time. When the workload increases beyond what the server is able to absorb on the other hand, I expect the end-to-end latency to increase, at least statistically, i.e., some responses will take more time to be processed by the server. My methodology is designed to probe an application server in order to identify the workloads at which the corresponding latency increases statistically, which I call *transition state*[1].

My server probing algorithm sends requests to the server, at varying request rates (called ***test throughput*** and denoted by $T_{test}$). At the same time, I timestamp requests and responses to obtain the end-to-end ***observed latency***. As only statistically significant variations in the observed latency are relevant, each value of the $T_{test}$ is probed for a few seconds. This period of time during which the $T_{test}$ is constant is called a ***sample window***. The reason for selecting a time window is to avoid wrongly inferring that the server is overloaded as the result of artefacts in the measurements, e.g., if the network path latency increases for a very short period of time without the server being actually overloaded. Sample window should not be too short or too long as being too short would be affected by temporary artefacts in measurement whereas being too long would increase the runtime of the measurement. In out measurement a sample window between 8-10 seconds results a more stable output.

Roughly speaking, my strategy is to go through increasing workloads, until I reach one for which the server shows signs of being overloaded, observed through increases in end-to-end latency.

[1]The rationale for this terminology is that the application server behaviour changes quantitatively during the transition state.

My design is separated into two phases: 1)***Fast ramp-up***: increasing the $T_{test}$ exponentially until the observed latency shows signs that the server is getting overloaded; 2) ***Fine-tuning***: roll back and converge slowly towards a value of the $T_{test}$ in the transition state where the observed latency meets the user defined latency threshold.

### 3.1.2 Design

In both phases a given $T_{test}$ is generated, and the observed latency is measured. A pseudocode overview of my strategy is depicted in Algorithm 1. During the ***Fast ramp-up*** phase, the algorithm searches for a value of $T_{test}$ at which the latency starts increasing. So The Fast ramp-up phase starts with an initial $T_{test}$, with subsequent sample windows exponentially increasing the $T_{test}$ as long as the current window is marked as "accepted".

Initial $T_{test}$ is an arbitrary value. My algorithm is robust enough to reach to an estimation as long as the initial $T_{test}$ is not greater than the estimated throughput. This is because a very high value of $T_{test}$ might end up with a very high measured latency value at the beginning which results the algorithm not to increase the $T_{test}$ at all and not to return any estimation value at the end. Where the initial $T_{test}$ is not known, it is set to 1 request. The downside of defining 1 request at the beginning is that it makes the runtime longer since the fast ramp up phase takes more time to reach to the overload state of application. Likewise during my first experiment starting with the initial $T_{test}$ of 1, I launched the node with the lowest available resource offered by cloud providers. The estimation for that run was about 1.2K requests per second. Therefore I set my initial $T_{test}$ value to 800 requests slightly less than the lowest estimation. This value was my default initial $T_{test}$ for any further experiments on the other nodes. Therefore, my recommendation is to start with less than 1000 requests in cloud environment.

In this thesis, I are interested in the server-side latency only. However, the network delay component of the end-to-end latency might inflate the end-to-end latency and mislead us to believe that it is a server-side problem. To mitigate such occurrences, I also measure the network latency (RTT) using tcpping [tpi]. If the measured network latency during a sample window increases significantly(more than 20ms) compared to the baseline(tcpping Round Trip

Time value), I discard the measurements from the corresponding sample window and start new measurements of the sample window(RTTChanges function at line no. 11 in Algorithm 1).

If the median end-to-end latency is close enough to the measured RTT(not greater than 10ms), i.e., meaning that the server-side latency is very small, then I assume that the server is able to handle the current $T_{test}$. The reason behind picking the median is that I consider the majority and proceed based on that. I repeat that for each second of the sample window and if the condition is satisfied, this second will be tagged as "Pass". If more than half the seconds of the window are flagged as "Pass", the sample window is marked as "accepted" and the next $T_{test}$ to be probed within the next sample window will be increased (line no. 18 in Algorithm 1).

On the other hand, if a sample window is not marked as "accepted", it means that I have entered the transition state and with this, ***Fine-tuning*** phase will be started(line no. 20 in Algorithm 1). *Fine-tuning* phase of the algorithm tries to refine the value of the $T_{test}$ to get close to the true value of $T_{test}$ with which the server gets overloaded. In this phase, I increase the $T_{test}$ linearly instead of exponentially, starting from the highest $T_{test}$ that did not show signs of latency increase, i.e., the $T_{test}$ from the previously accepted sample window(line no. 22 in Algorithm 1). In this phase, the increase in $T_{test}$ between consecutive sample windows is a fixed rate of the gap between the last two $T_{test}$ during the ramp-up phase(line no. 23 in Algorithm 1). The exact rate used defines the degree of fine-tuning for this phase of the algorithm. Picking the value of rate results in a trade off between runtime and the granularity of the output, i.e., if a small rate selected, the value of increase in $T_{test}$ would be small. Therefore, the output value would be more accurate and close to actual value that the server can sustain whereas the run time of the algorithm ends up being high as a result.

At the end of the fine-tuning phase, I will have sampled different $T_{test}$ values, the last one being the one that my algorithm stops at, which I call $T_{estimate}$. As stopping criterion for the fine-tuning phase, I rely on a percentile and latency threshold of the response time which are defined at the start . The reason for this lies in the common use of response time percentiles for SLA's [CP09].

If the $n^{th}$ percentile response time of a $T_{test}$ crosses the user defined *SLA latency threshold* dur-

```
1  func probServer(T_test){
                                                    /* Probe the server with T_test  */
         /* Return a list of accept or reject values for each second and measured
   latency values */
2  }
3  func RTTChanges(){
          /* Probe and measure network latency using tcapping to identify network
   instability  */
                                              /* Return True if network is unstable  */
4  }
5  SLA = 'User_Defined_Value'
6  Percentile = 'User_Defined_Value'/* Usually 90th percentie  */
7  Phase = 'FastRampUp';
8  T_test = User_Defined_Value/* ops/sec , choose 1 if you don't have any idea */;
9  while TRUE do
10 |  SampleWindow_list = probeServer(T_test);
11 |  if RTTChanges() then
12 |  |  continue;                  /* Continue probing as network is not stable  */
13 |  end
14 |  else
15 |  |  switch Phase do
16 |  |  |  case ('FastRampUp')
17 |  |  |  |  if median(SampleWindow_list) == 'Accept' then
18 |  |  |  |  |  T_test=T_test * 2;                  /* Ramp-up - exponential increase */
19 |  |  |  |  end
20 |  |  |  |  else                                          /* Overloaded_server -  */
21 |  |  |  |  |  Phase = 'FineTuning'
22 |  |  |  |  |  T_test = T_test/2;
23 |  |  |  |  |  inc = rate * (T_test/2);
24 |  |  |  |  end
25 |  |  |  case (FineTuning)
26 |  |  |  |  if Percentile(SampleWindow_list, SLA) then
27 |  |  |  |  |  T_test = T_test + inc;                          /* Linear increase */
28 |  |  |  |  end
29 |  |  |  |  else                                         /* Found value of T_test */
30 |  |  |  |  |  return T_test − inc;                /* Return estimated throughput */
31 |  |  |  |  end
32 |  |  endsw
33 |  end
34 end
```

**Algorithm 1:** Pseudo-code of the methodology.

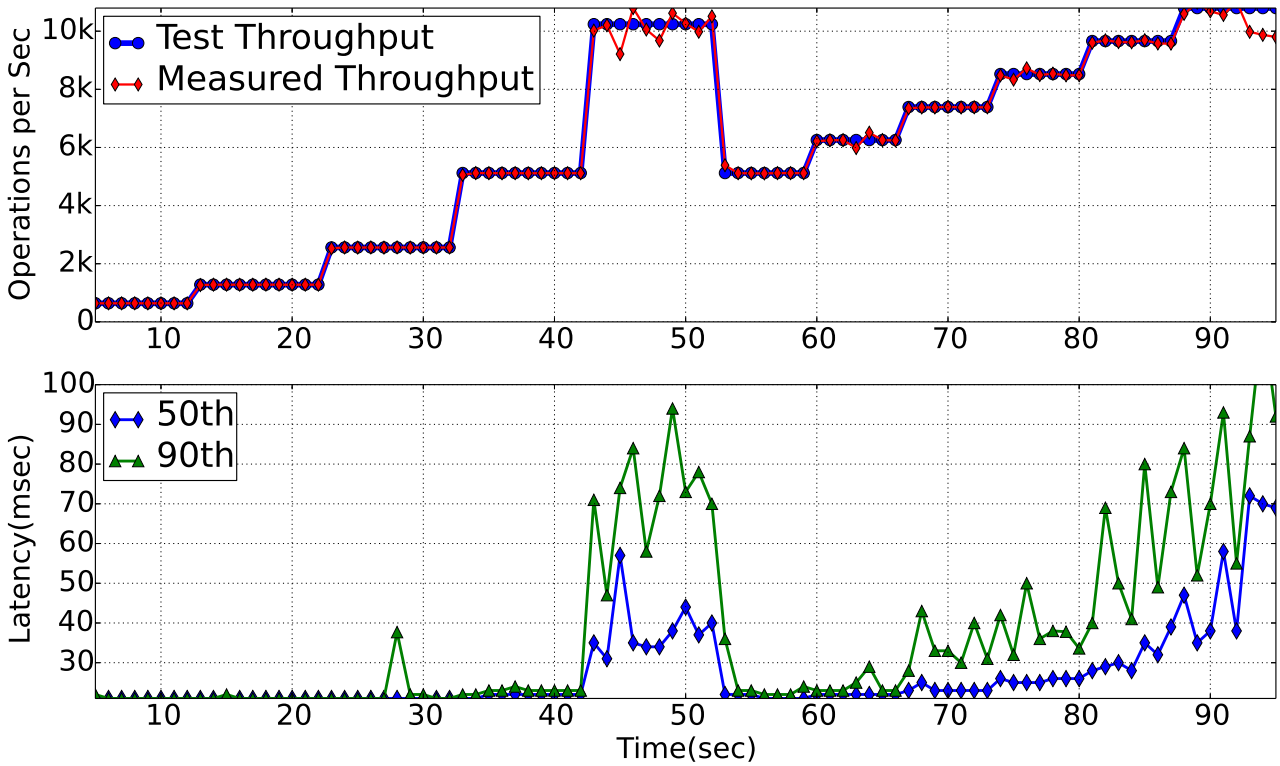



Figure 3.1: Time series of a run of the methodology on a Cassandra server deployed in GCE.

ing the fine-tuning phase, the algorithm terminates and the $T_{test}$ from the previously accepted sample window is returned as $T_{estimate}$. I use $T_{estimate}$ in Chapter 4.3 to evaluate application performance on various cloud platforms. The specific SLA settings (percentile and latency threshold) are configurable parameters of my tool, as different applications require various levels of responsiveness. In my experiments, I use as SLA latency the $90^{\text{th}}$ percentile and 150ms.

### 3.1.3 Example

To illustrate my methodology, I present in Fig. 3.1 the results from a single run of the algorithm using a server located in the Google Compute Engine cloud platform [gce14]. The top plot of Fig 3.1 shows the values of the $T_{test}$ (in operations per second), as well as the measured throughput (from the client side).

I observe that the fast ramp-up phase ends at second 52(the exponential increase), followed

by the fine-tuning phase(the linear increase). I can also see on the top plot of Fig 3.1 the exponential increase in the values of the $T_{test}$ during the fast ramp-up phase, and the linear increase during the fine-tuning phase.

The lower plot of Fig. 3.1 shows the $50^{\text{th}}$ and $90^{\text{th}}$ percentiles of the end-to-end latency over time. The median values of RTTs from the client to this server was about 11ms. I observe that the latency percentile values are in the same range as the RTT during the fast ramp-up phase(blue and green lines are at the range of 13ms), until the last sample window of the phase (between seconds 42 and 52). At that sample window, the 50th and 90th percentile values jump to high values(over 30ms). This is the sign for the algorithm to reduce the $T_{test}$ value to the previous sample window(5200 requests per second). The second part of the algorithm (fine-tuning phase) starts (between second 52 and 95) and the $T_{test}$ values are increase linearly based on the defined rate(0.2(20%) * 5200). The transition happens at throughput values between 5k and 10k operations per second. I observe, as expected, that the lower percentiles of the latency are less sensitive to the changes of $T_{test}$.

### 3.1.4 Discussion

I have designed my methodology towards estimating the throughput that a server can absorb without significantly increasing the end-to-end latency. Therefore, the final look of the algorithm is strongly dependent on my initial goal. Nevertheless, I have tried to limit my assumptions about the application under test, and the server behaviour, making it generic enough to be applicable to many applications and server platforms.

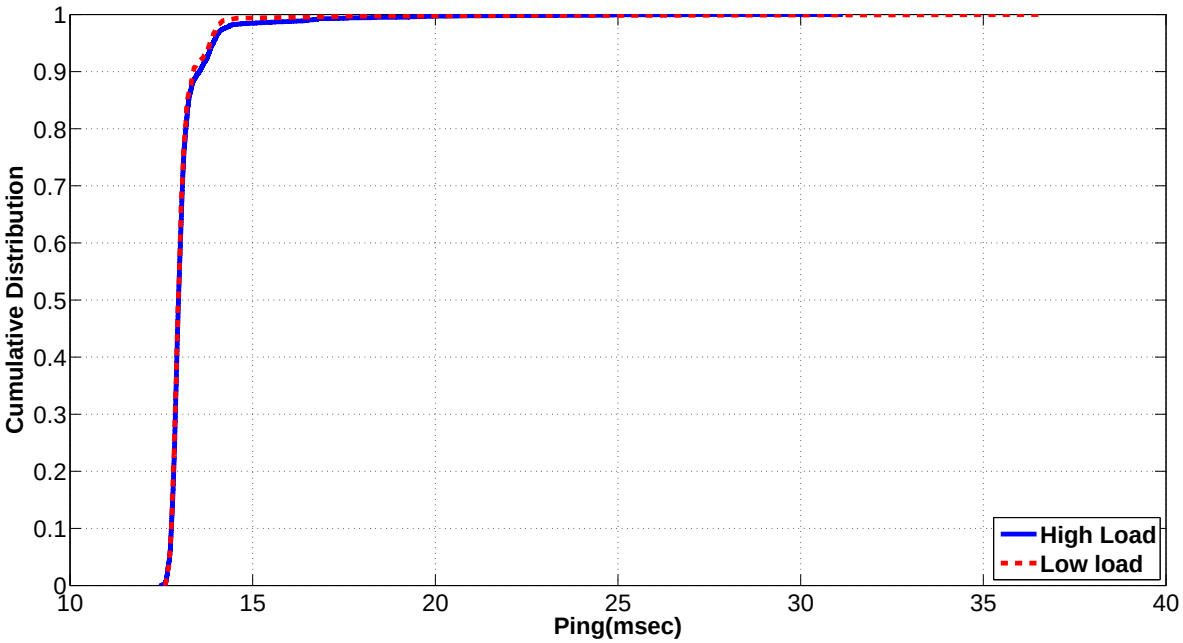
During the design of the algorithm, I had to choose the ramp-up policies in each phase, and these policies affect the granularity of the $T_{test}$. I have selected those policies with the goal of going fast enough through the $T_{test}$ values, to find the one at which the latency starts increasing. The exponentially increasing fast ramp-up phase intends to find a rough approximation of the throughput range of the phase transition, while the linearly increasing fine tuning phase narrows down to a finer throughput region around this phase transition. While other ways are possible, e.g., bisection-based approach for the fine tuning phase I believe that my current

design leads to a reasonable compromise between speed and accuracy of the $T_{test}$ estimation. Methods like bisection-based approach reduce the runtime of the algorithm, however I would miss the trade off between measured latency and $T_{test}$ values in fine tuning phase (I will illustrate the benefit in section 4.4).

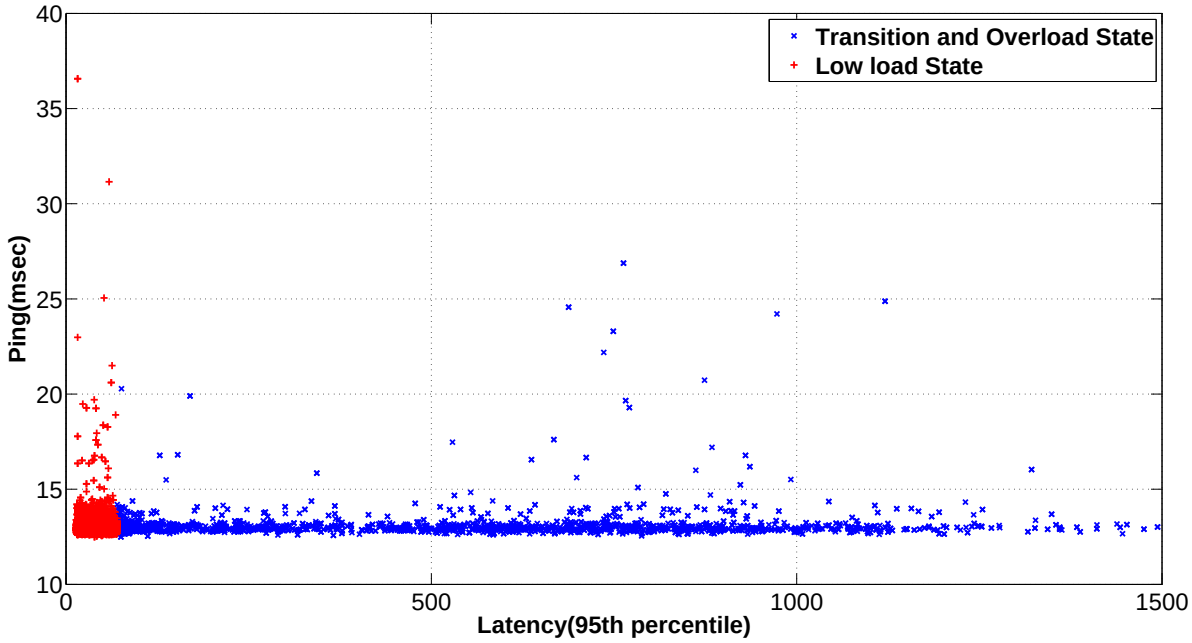
As already explained, I measure the RTTs and compare them with the end-to-end latency to decide about the increase of the next $T_{test}$. Moreover, Tcpping values that are captured at the end of each sample window are monitored and verified so that RTTs are not affected with low and high volume of load. Figure 3.2a shows the cumulative distribution of tcpping values when I ran the benchmark for about 24 hours at both low and high loads traffic. It shows that in both cases the variation is low and RTTs has not been affected by high load.

It is possible that the RTTs may spike during the measurements, e.g., due to routing instabilities or congestion on the path. Figure 3.2b shows the scatter plot of the tcpping values of same previous measurements, very few (1.45% out of total) high values of RTT spikes (more than 15msec) are observed but my methodology has been designed to be robust enough not to be affected by this issue. Indeed, if a statistically significant change of the measured RTTs(20ms increase over RTT) for half of the seconds across sample windows is observed, I discard the measurements from the suspicious window and redo the same $T_{test}$. Shorter and burstier network latency changes are discarded implicitly thanks to the sample window. Smarter approaches are possible, e.g., by trying to identify the exact time when the change in RTT happened and removing the corresponding bias in the end-to-end latency. However, given that many routing events are transient in nature and likely affect the RTTs for relatively short periods of time [PZMH07], I believe that for my current purpose, the added complexity is not worth the effort.

Finally to ensure better sampling of throughputs and latencies I recommend replaying the methodology using different values of initial $T_{test}$. Different initial $T_{test}$ results in testing more ranges of $T_{test}$ and corresponding latency values.




(a) CDF of tcpping values at high and low load




(b) Scatter plot of tcpping values vs. 95th percentile end-to-end latency repeated every hour for about 24 hours in medium size instance in EC2.

Figure 3.2: CDF and Scatter plot of Tcpping values over 24 hours

## 3.2 Summary

In this chapter, I have presented a black-box technique that measures the performance of cloud applications. I probe the application remotely, iteratively adjusting the generated load based on the measured latency from previous steps. Using my technique, I will be able to estimate the maximum capacity of an application for a given SLA. This allows us to compare the application performance within a cloud provider offering various types of nodes with various resources in the next chapter.

# Chapter 4

# Throughput Benchmarking in the Cloud

Public clouds are most likely the first option for small or medium size application providers who want to either deploy new applications or migrate their existing applications to cloud. There exist many public providers that offer various on-demand virtual machines with various types of resources (e.g., AmazonEC2 offers about 25 different types of instances). In this chapter I have implemented the methodology explained in previous chapter and use my measurement platform to benchmark an application on various types of VMs using different cloud platforms. This will allow an application provider to estimate the performance of an application with respect to predefine latency when it is deployed on any VM type within a cloud provider or across different cloud providers.

## 4.1 Building A Measurement Platform

To perform my measurements I have designed and implemented a measurement platform. It is a client server application that consist of a multi-thread server side program that I called "controller" and a client side program that is implemented on planetlab nodes across the world. The controller talks to the clients using JSON-RPC [jso14]. JSON-RPC is a remote procedure

call protocol encoded in JSON. The controller manages the experiments by defining a test plan. A test plan is a json object consists of 1) Name and ip of Cassandra server that has to be benchmarked 2) Name and ip address of the planetlab node that runs the test 3) number of runs 4) Time schedule for run. The controller creates one thread per client and calls a method on the remote clients (planetlab nodes) by sending the JSON test plan object. Once the controller sends the test plan, each of the planetlab nodes involved in experiment starts its own experiments based on the details of its own test plan. Finally, when each client finishes its experiments, it will return the results to the controller for further analysis. The measurement platform provides the capability of running multiple experiments at the same time. It is also deployed across many nodes on the planetlab platform. I have also added the capability of grouping the nodes, e.g., based on continent so that nodes from a particular continent will carry out assigned experiments. Figure 4.1 shows the measurement platform and the entities that are involved in this platform. The application servers (Cassandra in my experiments) can be deployed in any available public cloud, e.g., Amazon AWS, Microsoft Azure or Google Compute Engine (GCE).

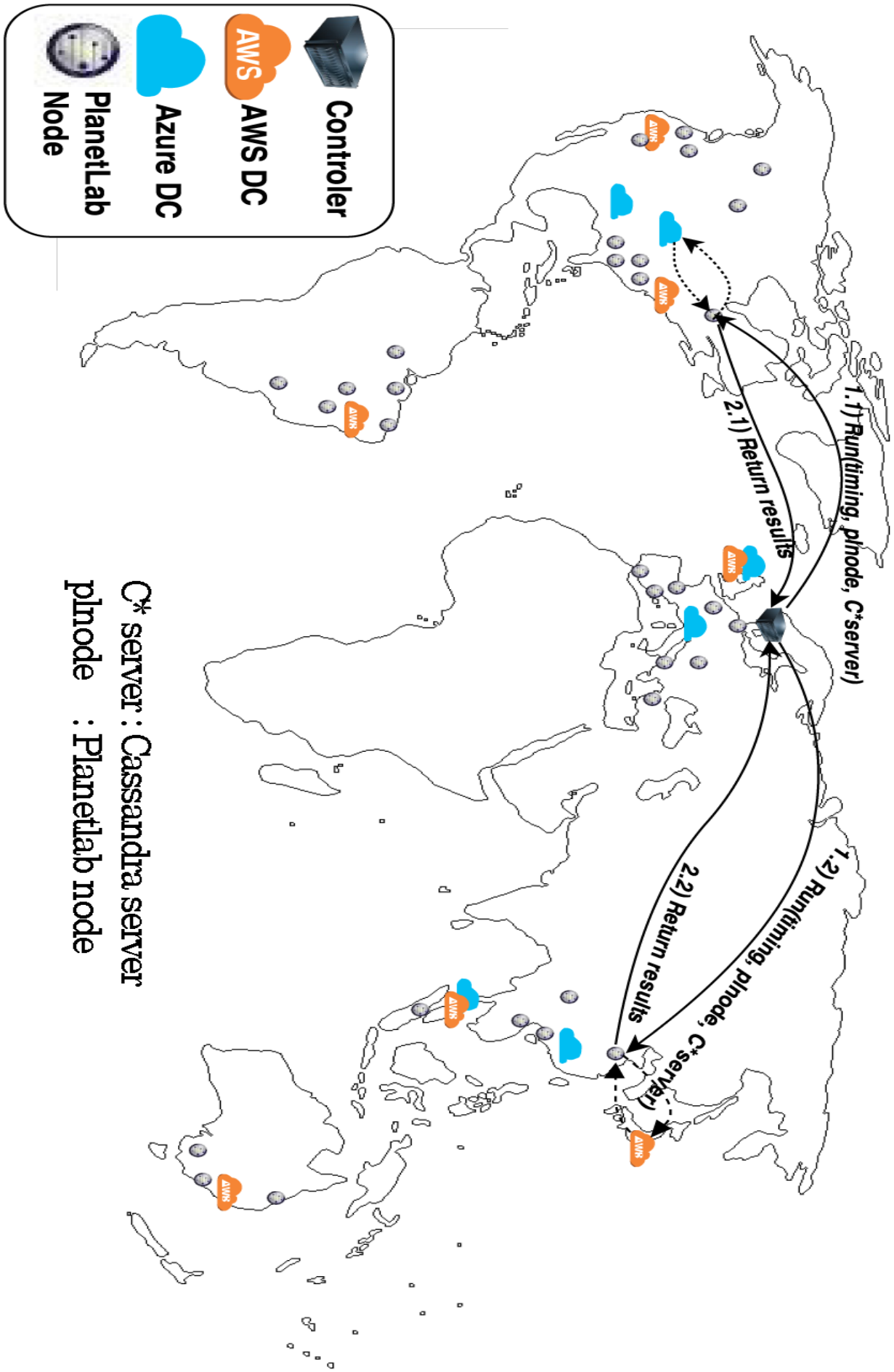



Figure 4.1: A view of the measurement platform

## 4.2 Methodology Implementation

To Implement the proposed methodology, the first part was either writing a workload generator from the scratch or finding an available workload generator. I found two benchmarking tools that both have a workload generator and are suitable for my purpose: 1)Apache JMeter [jme14a], an open-source tool to load test functional behaviour and measure performance. It was originally designed for testing Web Applications but has since expanded to support a verity of other applications. JMeter architecture is based on plugins. Most of its "out of the box" features are implemented with plugins. Off-site developers can easily extend JMeter with custom plugins. 2)YCSB[CST+10], an open-source program suite for evaluating retrieval and maintenance capabilities of datastores. It is often used to compare relative performance of NoSQL database management systems.

The above tools offer benchmarking applications i.e., they will extremely overload the application and return a number as a benchmarking score for that application. Unlike the existing benchmarking applications, my proposed framework is designed to provide each estimation based on a defined SLA latency value. In fact my tool is built on top of such benchmarking applications to benefit from their load generation capability but in line with my requirement which is taking into account the usage of defined SLA latency value for any estimation(e.g., Figure 4.2).

Eventually, the methodology was implemented as a plugin for Apache JMeter. There was two reasons why JMeter was chosen over YCSB. 1) Load instability was the main reason that I moved on from YCSB. YCSB is not usually capable of generating the exact number of requests in each second. This problem is intensified when it compensates the shortcoming load for a particular second to the subsequent second. For instance, when at second X the load is designed to be 100 while the actual number of generated requests is 80, YCSB will carry 20 requests to the next subsequent second. This effect will mislead the measurements by increasing the measured latency values which is the fundamental part of my methodology. 2) YCSB is designed specifically to benchmark datastores, however JMeter offers benchmarking datastores as well as other types of applications. In section 4.6 it is shown how my designed tool works

together with JMeter to estimate the throughput of a stateless web server. Being free of such obstacles, JMeter was assuring enough to be chosen for implantation of my methodology.

I have implemented the methodology described in Chapter 3.1 as a plugin for Apache JMeter [jme14a] (tested with v2.11). The architecture of JMeter and my implementation is depicted in Figure 4.2. The lowest part is the core part of JMeter which manages threads, generates and controls the load.

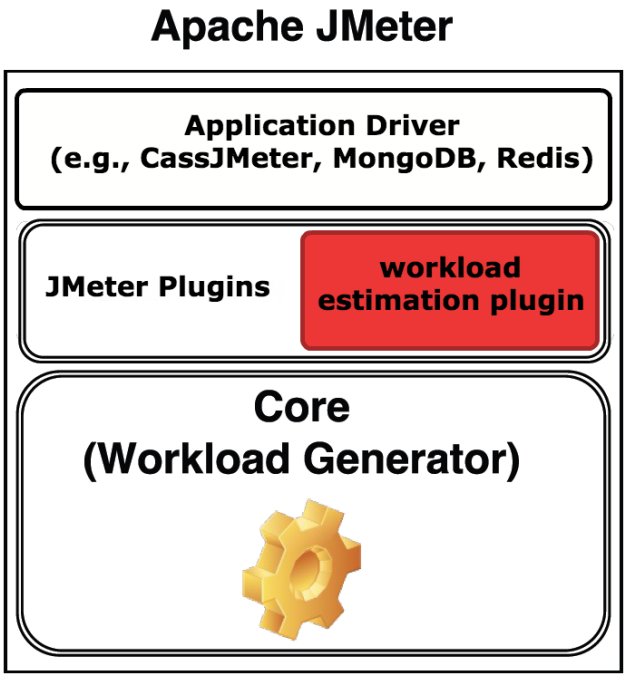



Figure 4.2: Workload Estimation System Design

JMeter has also been designed with a highly extensible core, meaning that new samplers, timers and visualisation functionality can be plugged into the core system to extend its capabilities. My workload estimation tool is designed to be part of the JMeter-plugins standard set [jme14b]. This extensibility helps to attach any applications but only needs to integrate the driver of application with JMeter. The top part is composed of the interface drivers that integrate the target application with core JMeter. My workload estimation plugin will work independent of application as it is injected in plugin part. For my purpose since I was interested in datastore applications, I have tested Apache Cassandra deployments [LM10] using the CassJMeter [cas14] 0.2 interface driver for JMeter integration.

Apache Cassandra is a distributed key-value storage system for managing large amounts of

data potentially partitioned and replicated across multiple servers. This type of application provides on-line read/write access to data in web. Usually when a web user is waiting for a web page to load, reads and writes to the database are carried out as part of the page construction and delivery. In all my experiments I consider the simplest operation to be run on each node. Therefore, I perform read operation for a specific row in database. My plan is to open source the tool, so that others may use and extend the tool, and contribute new workloads and database interfaces.

## 4.3 Throughput Benchmarking

I now use the methodology explained in 3.1 and following the implementation of my tool explained in 4.2 , I obtain a $T_{estimate}$ for each run. This will be used as a metric to evaluate the performance of a Cassandra node using various types of VMs in different platforms, ranging from PlanetLab to well-known public Cloud providers.

#### PlanetLab

**Planetlab** [CCR+03] has been used by researchers for more than a decade for network and distributed services experimentation. With PlanetLab, each user receives a slice equivalent to a virtual machine. Experiments can be run on the slice without having any control on the underlying hardware and network infrastructure. Therefore, the performance of applications deployed in this platform is highly variable. I expected concurrent experiments, and hardware heterogeneity to drive the observed performance in this platform.

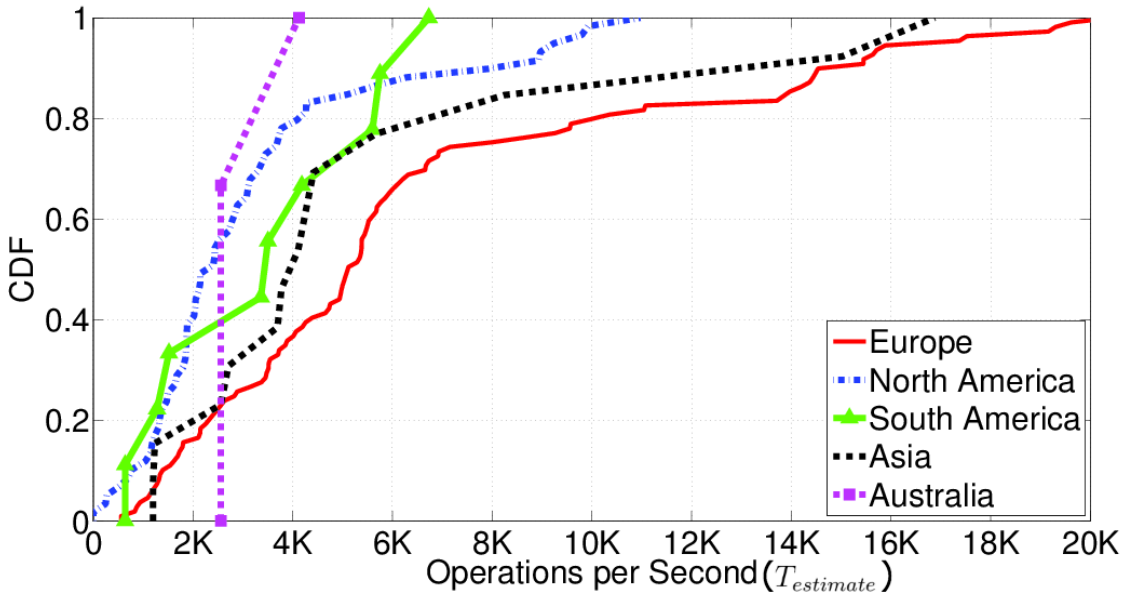



Figure 4.3: $T_{estimate}$ measured for 193 Planetlab nodes: 109, 59, 9, 13, 3 nodes in Europe, North and South America, Asia and Australia, respectively

Figure 4.3 shows the distribution of $T_{estimate}$ values for a worldwide sample of 193 nodes. I have split the nodes across continents to ease the comparison. The CDFs illustrate the broad spectrum of nodes with different performance ranges across the platform. Surprisingly, some PlanetLab nodes have a performance equivalent to a high-end node in a public cloud platform, e.g., a "large" node in EC2 or a "standard2" in GCE. In overall better performance for European nodes is observed, most likely because the nodes are added more recently and have more resources compared to old nodes.

### Public Cloud platforms

Public cloud platforms typically span multiple geographic regions around the world. Each region contains several availability zones (AZs) that are physically isolated and have independent failure probabilities. One AZ is roughly equivalent to one data center. A VM in such platform is called an instance. Different types of instances come with different performance characteristics and price tags.

**Microsoft Azure**. I chose three types of instances among the options offered by Azure (small, medium and large with 1, 2 and 4 cores and 1.75GB, 3.5GB and 7GB memory, respectively). For

each instance type, I deployed a Cassandra node in 6 different AZs (2 in US, 2 in Europe, 2 in Asia). Figure 4.4 shows the box-plot diagram of the returned throughput for each combination of location and instance type. The node is created and populated with one record. Then, the memtable[1] is cleared to force Cassandra to read the record from disk. Experiments ran for a 24 hour period during which I performed a run every 15 minutes. The node has not been stopped during the experiment. Figure 4.4 shows a clear differentiation of the $T_{estimate}$ for different types of instances. Within the same instance type, different locations showcase varying levels of performance. I expect that one of the main factors explaining the measured performance of different instances is the type of hardware used, as well as the level of user multiplexing on the sampled blades.

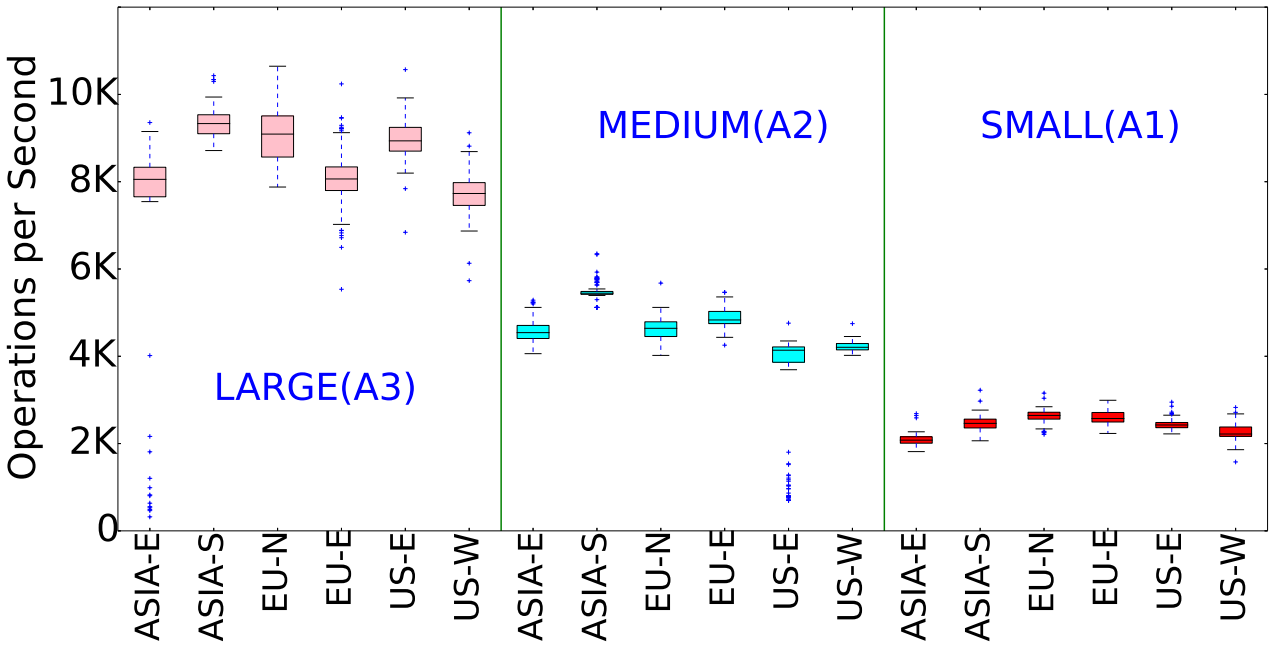



Figure 4.4: Benchmarking three types of instances across different datacenters in Microsoft Azure.

I also observed performance variations on equivalent instances at the same location. In particular, the figure shows a temporary performance degradation observed in two different instance types (low throughput data points between 0-2K operations per second), namely the "Large" instance in Asia East (ASIA-E inside LARGE (A3) in Fig 4.4) and the "Medium" instance in US East (US-E inside MEDIUM(A2)). My observations show that for a short period of time, a sudden dip in performance happens, followed by a gradual recovery.

[1]Please refer to Appendix A for further details about how Cassandra reads and writes the records.

**Amazon EC2**. I now look at the results observed on the Amazon EC2 platform. I chose t.micro, m1.small, m1.medium, m1.large, m1.xlarge instances with 2vCPU, 2ECUs/1VCPU, 2ECUs/1VCPU, 4ECUs/2VCPU, 4vCPU and 1GB, 1.7GB, 3.7GB and 7.5GB 15GB memory, respectively. Figure 4.5 left side shows the $T_{estimate}$ values achieved from five instance types for one AZ in Europe.

Measurement shows that my methodology is able to translate the nodes with various resources to a meaningful value that is recognized at the application level. The $T_{estimate}$ output value can be a metric for comparing performance of arbitrary selected nodes from a pool of available resources in cloud. Result shows that a micro instance shows better performance than a small or medium instance. The reason is that the micro (t series) instances support burstable performance ,i.e., they allow utilizing 10% of a full core of a CPU (called baseline level of CPU and counted as credit balance if 10% usage has not utilized). If at any moment the instance does not use the 10%, it stores them in its CPU Credit balance for up to 24 hours. Therefore, when the t1.micro needs to burst to more than 10% of a core, it draws from its CPU Credit balance to handle this surge seamlessly.

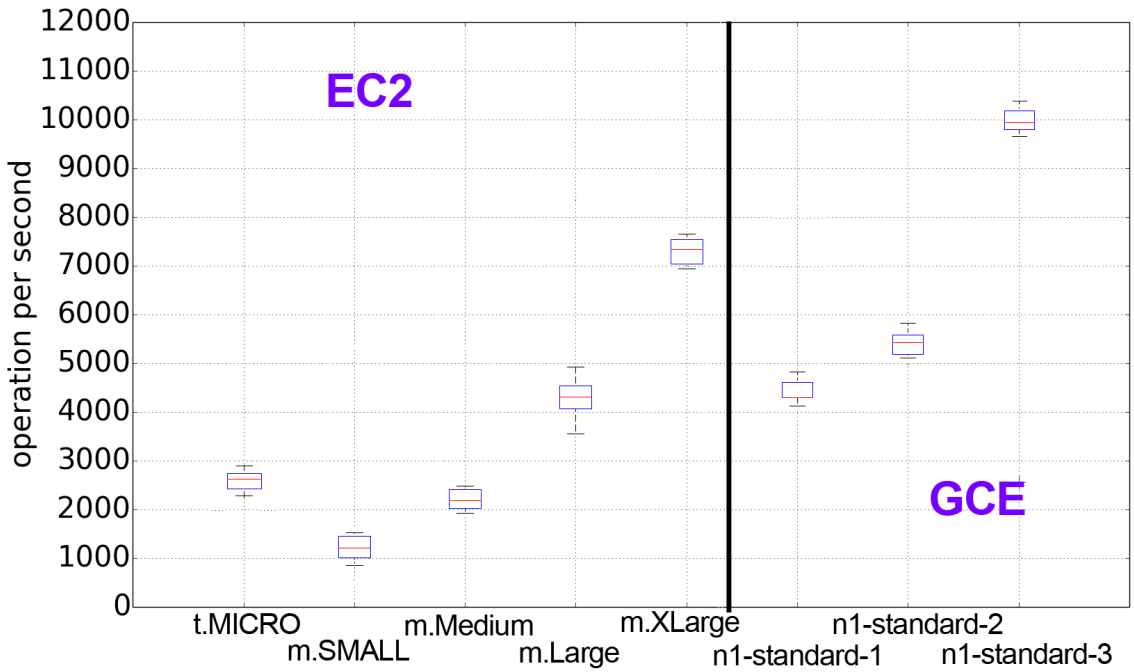



Figure 4.5: $T_{estimate}$ for different types of instances located at Europe's datacenters in two public cloud platforms namely Amazon AWS and Google Compute Engine (GCE).

**Google Compute Engine (GCE).** I provide the result of running the benchmark on Google

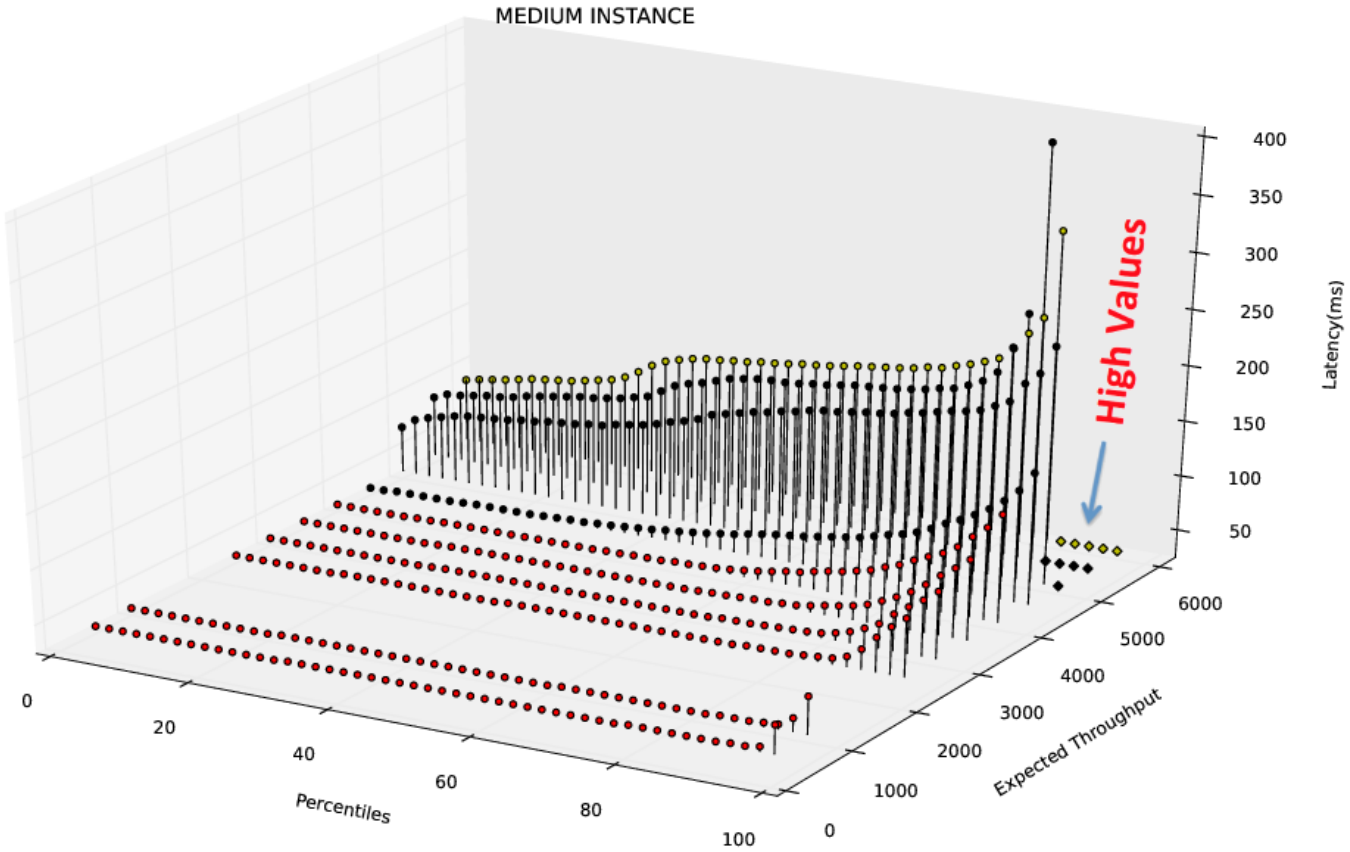
compute Engine platform which I have obtained by running the benchmark on different instances in west Europe 'zone b' including n1.standard-1, n1.standard-2, n1.standard-3 with 1VCPU with 3.7GB, 2VCPU with 7.5GB, 4VCPU with 15GB respectively. Figure 4.5 (right) depicts $T_{estimate}$ values for each type of instance.

Overall, the results illustrate that my methodology is useful for cloud users and can help them to observe performance of application for a given instance in cloud before deciding about choosing the nodes to deploy the application. Measurements show variation in performance and the capability of my proposed methodology to detect that. I believe that my methodology is useful to expose nodes with poor performance, triggering their redeployment to a fresh VM that exhibits the expected performance.

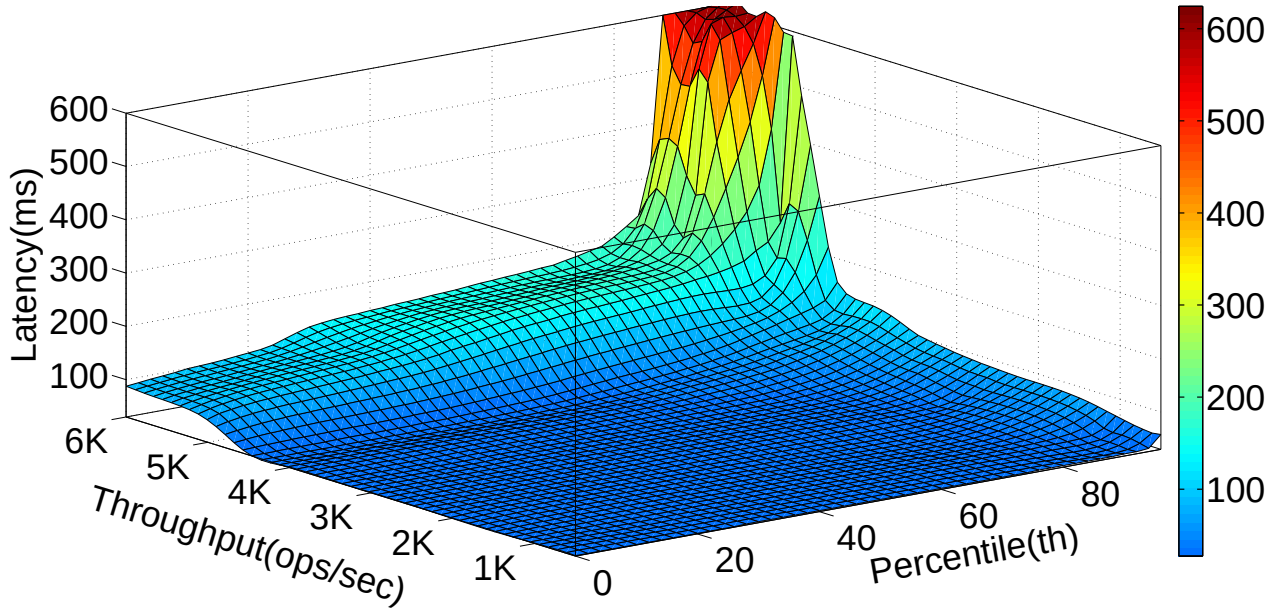
## 4.4 Latency/Throughput Trade-off

I have shown in the previous section the estimation of the maximum load an application server can sustain, under a predefined latency SLA. While valuable, this metric only reflects a partial view of reality. Application providers might also want to know how a server will behave in the presence of various workloads. I can also use my methodology to understand this aspect of application-level performance. In this section I further explore the relationship between throughput and latency.

My methodology by design, samples multiple values of the $T_{test}$, and for each value it records the value of different end-to-end latency percentiles. I have picked the output of one run of benchmark for two different types of instances. I note the observed latency measurements and calculate different percentiles of latency values ranging from 2 to 99. I plot 3d figure using values of percentile number, corresponding latency and throughput. Figure 4.7a and 4.6a depict the scatter plots of the observed median latency at different $T_{test}$ for large and medium instances in Microsoft Azure public cloud platform. Note that for better readability of the graphs, the very high values at the very right corner are plotted at zero value(marked as high values) for better visibility and to avoid mixing with the other data points. Figure 4.7b and 4.6b are the

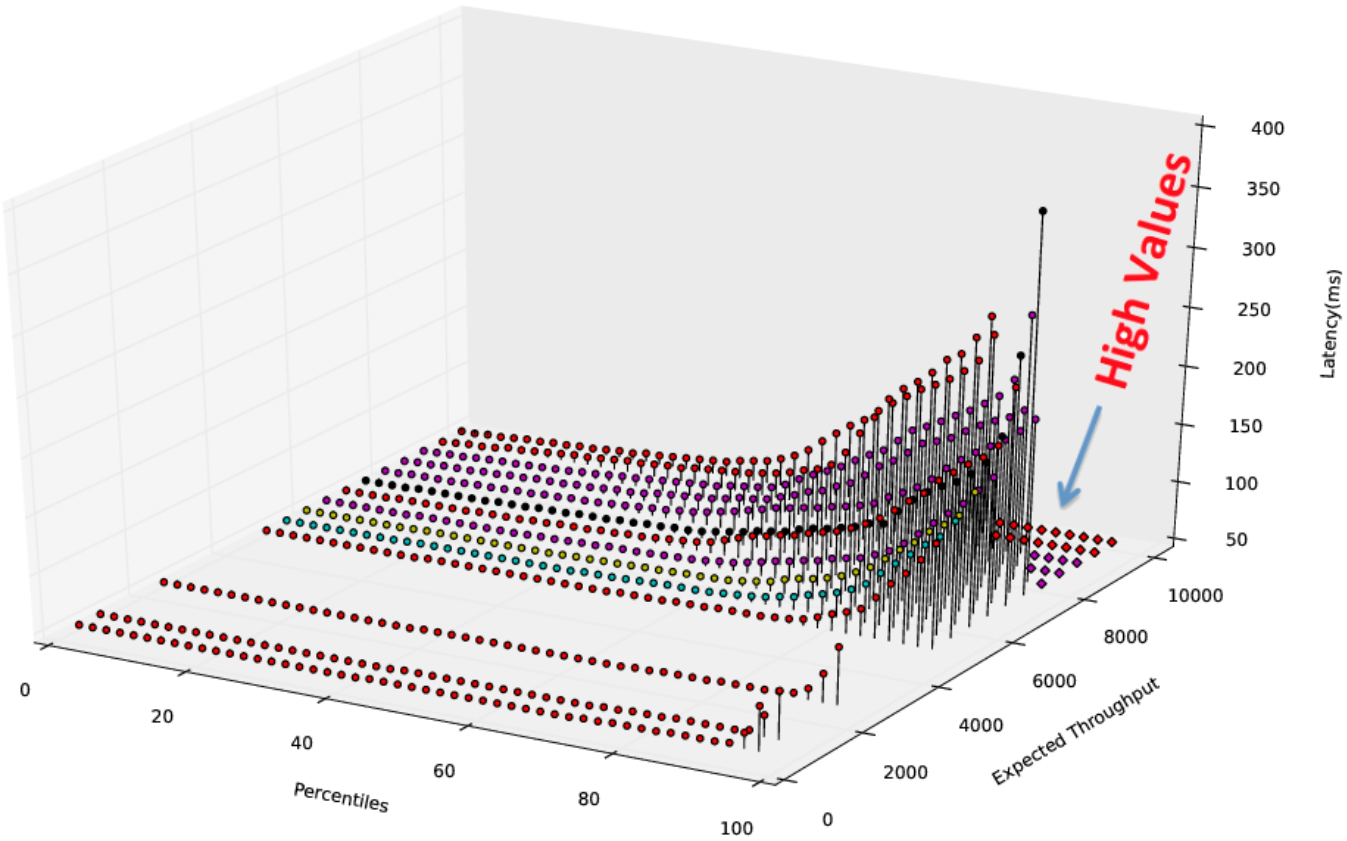



(a) 3d scatter plot of different throughputs and corresponding latency percentiles

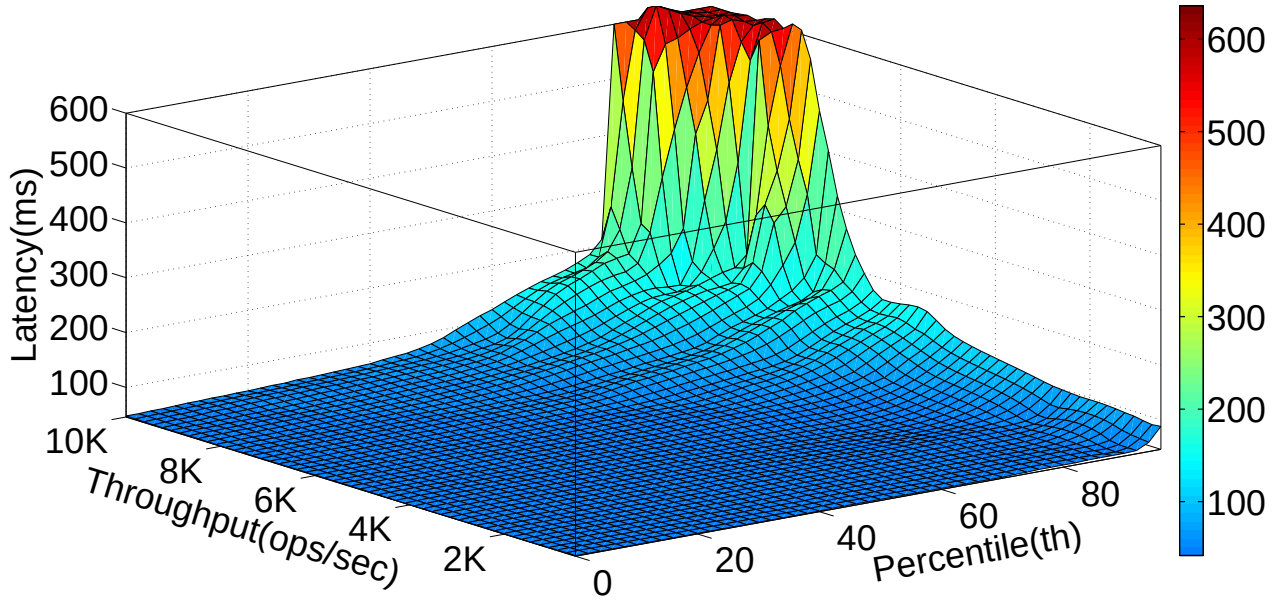



(b) Surface plot of different throughputs and corresponding latency percentiles

Figure 4.6: Latency/Throughput trade-off for a **medium** instance (A2) in Microsoft Azure




(a) 3d scatter plot of different throughputs and corresponding latency percentiles




(b) Surface plot of different throughputs and corresponding latency percentiles

Figure 4.7: Latency/Throughput trade-off for a **large** instance (A3) in Microsoft Azure

corresponding surface planes which interpolate the median latency values that are observed for the various $T_{test}$. As explained before throughout the experiment I store all the latency values at each percentile corresponding to different $T_{test}$. This plot can be generated after one run of experiment. Note that the more the throughput increases the more the corresponding median end-to-end latency values increase (in particular higher latency percentiles are more sensitive). The surface shows that the transition state is abrupt beyond certain $T_{test}$ in each case, i.e., as soon as the $T_{test}$ reaches values close to the overloaded state of the server, latency increases significantly for the high percentiles (e.g., typically the $75^{\text{th}}$ and higher). Note that similar node instances tend to have similar latency/throughput surfaces, with smaller instances (with less memory and computing power) displaying more abrupt changes at the transition state compared to larger instances. Moreover, the impact of the design of the ramp up policies is also reflected in Figure 4.7a and 4.6a. The fast ramp up tends to sparsely sample low throughput values, while the linear ramp up during the fine-tuning phase covers the higher loads extensively.

## 4.5 More Complex Workload and Throughput Estimation

While the assumption about the workload in the above sections is simple, i.e., one type of operation (read) over one record from disk, often application providers intend to test more complex workloads and observe the performance of an application in those scenarios. Based on such observations, application providers plan for deployment and provisioning VMs. Performance evaluation of an application depends on different elements. Figure 4.8 depicts different workload elements that each individually might affect the performance estimation of application (i.e., Throughput under certain SLA). For instance, each type of operation has its own functionality and system design and involves different parts of a system. In a read operation, data is fetched from a location on a disk, where it is already stored by a write operation. These operations usually involve memory in addition to disk for boosting the overall performance.

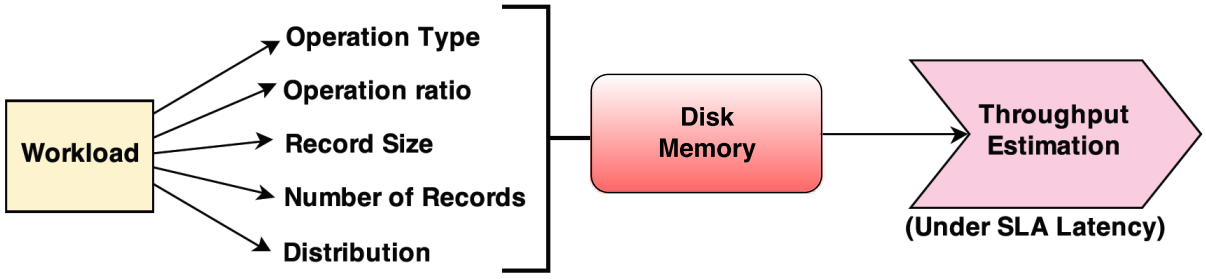



Figure 4.8: Workload elements involved in throughput estimation

Apart from the type of operation, size, number of records, workload distribution might also affect the throughput estimation. The way the records are being requested(access behaviour) decides if they should be fetched from memory or disk. Therefore, disk or memory access ultimately affects the overall throughput(one seek request to disk costs 10ms), e.g., some records are extremely popular and cached in memory and retrieved with a short response time.

### Type, Ratio, Distribution and Throughput Estimation

In this part, I generate different workloads with different operation types, ratios and distributions and show the throughput estimation in each case. I generate different workloads using both preliminary operations (read and write). I implemented multiple workloads described in [CST+10], selecting workloads representative of different types of applications. The records are requested according to both uniform and zipfian distributions (e.g., the Wikipedia articles are accessed according to a zipfian distribution [UPvS09]). In the case of zipfian when choosing records, some records are extremely popular while most records are unpopular. I utilized the implementation of zipfian used in [CST+10] and integrated it with my tool.

Table. 4.1 lists four different workloads that are considered as use-cases. I estimate the throughput for each case. Apache Cassandra involves both memory and disk to perform read and write operations.

Figure 4.9 depicts the estimations of throughput for each workload in Table 4.1 on a medium size node in the Amazon EC2 platform. I observe how the estimated performance changes de-

| Workload | Operations | Distribution | Application Example |
|---|---|---|---|
| W1-only Write | Write:100% | Uniform | A back-end to an Internet of Things (IOT) application keep writing the incoming data where data is fed into a BI system for further analysis |
| W2-only Read | Read:100% | Zipfian | User profile cache, where profiles are constructed elsewhere |
| W3-Read/Write | Read:95% Write:5% | Zipfian | Photo tagging; add a tag or update it, but most of the operations are read |
| W3-Read/Write | Read:50% Write:50% | Zipfian | Session store recording recent actions in a user session |

Table 4.1: List of Workloads

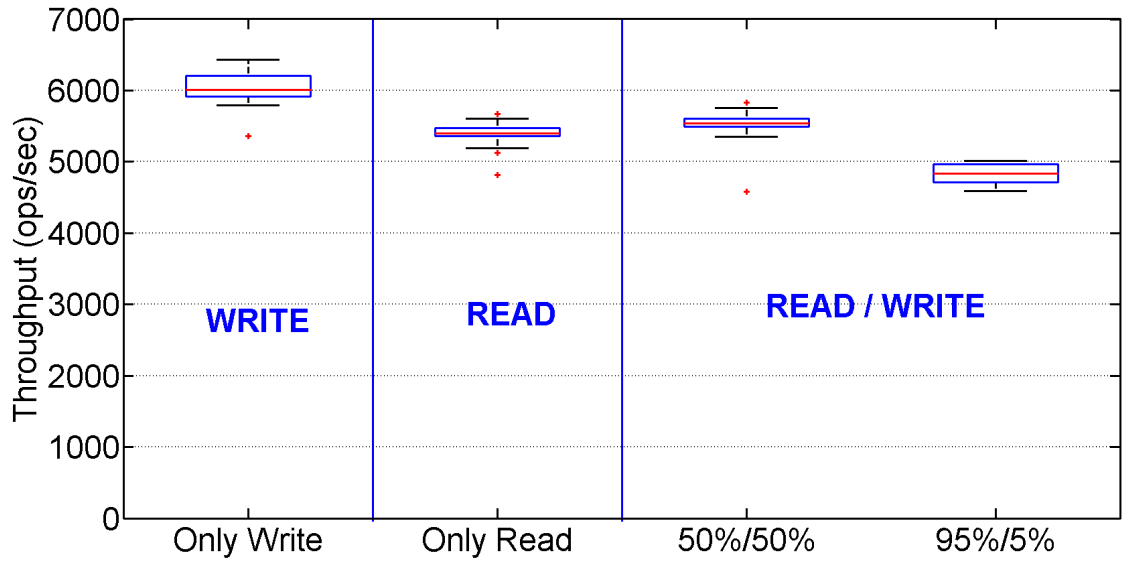



Figure 4.9: Throughput estimation of different workloads on a medium size VM in EC2

pending on the type of workload the application sustains, including the distribution of requests, and the ratio of read and write operations. Unlike lower-level benchmarks, my technique helps dimension the required server resources depending on the specific application scenario. The differences in performance between these scenarios come from the configuration of Apache Cassandra, and the hit rate of the different Cassandra cache systems to achieve higher throughput by attending requests in memory. High write performance is not surprising, that being one of the main design goals of this storage system.

**Size Impact on Throughput Estimation**

In this section I analyze the impact of varying the size on throughput estimation of read and write. I ran my tool and estimated the throughput by varying the size of records ranging from one character length (which is about one byte+100byte metadata) to 9Kbyte under SLA latency of 150msec for 90th percentile. Read and write operations are chosen at random and out of 1000 records for each different record size. Figure 4.10 shows boxplot of $T_{estimate}$ for read and write operations based on different record sizes ranging from 100byte to 9 Kbytes. The figure shows that in read, increasing the size of records has less impact on the throughput returned by my benchmarking tool. The reason behind it is that the sector size on disk is about 4k. So, increasing the size of records does not change the throughput as each time the one or maximum two sectors will be read. However, write performance in terms of the number will be degraded but in fact writing to disk happens less frequently so the overall throughput in terms of byte per second will grow. The purpose of this experiment is to show that my method is able to estimate the throughput for each type of operations and with different record sizes.

## 4.6 Throughput Estimation for a Stateless Web Server

To demonstrate that my methodology is general purpose and can be applied to a different application than a data store, I have deployed an apache web server and examined my technique. I sent HTTP GET requests to the Apache web server that was deployed on different types of cloud instances. When a HTTP request comes in, one child process gets the request and reads the request. Then it parses the URL, finds the file name corresponding to the URL, checks the file states, performs security checking, opens the file, reads its content and finally sends the content to the client. The child process then listens for the next request. I measured the latency values according to the methodology explained in Chapter 3. Using my technique I was able to estimate the maximum throughput that this particular application can sustain under a certain defined latency value. For my experiment the Apache web service was deployed on four types of instances using Microsoft Azure cloud (A0, A1, A2, A3 with shared, 1, 2 and 4 cores and 768, 1.75, 3.5 and 7GB memory respectively).

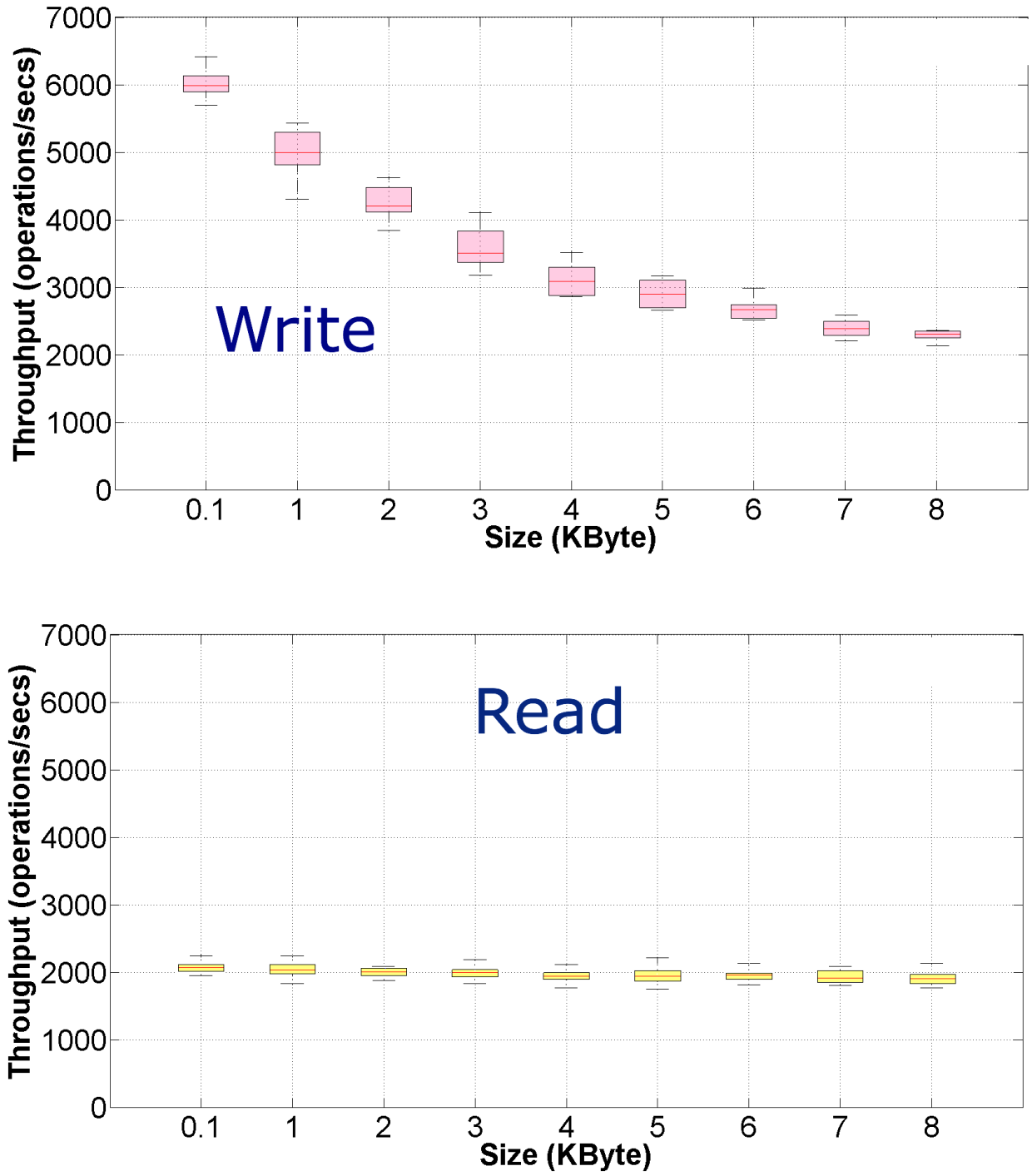



Figure 4.10: Throughput estimation of write and read when size vary in medium size instance in EC2

Figure 4.11 shows the box plot of $T_{estimate}$ values (HTTP GET requests per second) returned from each instance based on SLA latency of 100msec for 90th percentile. All the instances belong to the azure north Europe datacenter. As expected, the figure shows the nodes with higher resources handle more requests(high cpu, memory). Moreover, the results shows that

my methodology can be used effectively to identify the maximum throughput a web server can sustain despite the variety of available types of instance with different profiles in a heterogeneous environment like the cloud. In fact the throughput estimation results from the current and previous sections achieved for different applications show the genericity of my technique in estimating the throughput. In all cases I rely on latency as a sign to identify the maximum throughput regardless of what application is being examined.

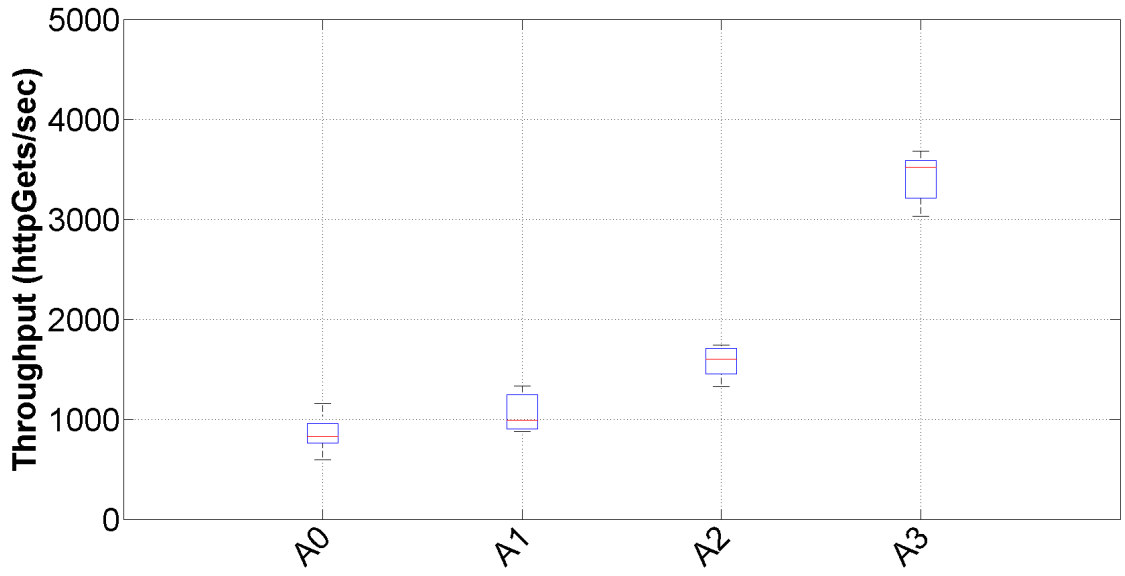



Figure 4.11: Estimated throughput (in terms of httpGet requests per seconds) of an Apache webserver deployed on four types of instance in Microsoft Azure

## 4.7 Summary

In this chapter I have implemented the methodology set out in the previous chapter and estimated the maximum load a cloud application can sustain on various existing types of VMs with different hardware specification in various cloud providers. This will translate an instance with certain profile (hardware specification) to a value of throughput which can be interpreted directly at the application level. I have also examined more complex workloads that involve different workload elements. Moreover, I have identified a trade-off between the throughput and latency of application servers. Finally, I have shown that my methodology can be used with another type of application.

# Chapter 5

# Future Work

In this thesis, I have proposed a methodology which provides a trade-off between application performance and server responsiveness on a given VM in cloud. One potential future area of research that can benefit from this thesis is the research towards controlling the response time by proposing techniques and algorithms to guarantee the response time of the application. For instance, in modern geo-replicated data-store applications [LM10], maintaining consistency[1] among replicas adds overhead to the system which will affect the performance of other replicas and results in rising the response time, particularly when the application requires strong consistency (meaning clients are intolerant to stale data) and the number of replicas is large. Understanding such scenarios is one potential direction for future work where my tool can help to analyse throughput latency trade-off in different cases e.g., one might setup application in multiple locations with different consistency level and run my tool to estimate the throughput in different situations. Another direction towards controlling the response time is design and implementation of an autoscale system which will add, move and remove the nodes to the current cluster in order to adjust the throughput output and maintain the response time of user requests under a certain threshold value. My proposed methodology and technique can help to understand the trade off between throughput and responsiveness of each replica. This can further lead to building a predictive model that proactively anticipate latency value based on

[1]Please refer to Appendix B for further details regarding replication and consistency policy in geo-distributed data store

observed throughputs. Such a prediction is at the core functionality of the autoscaling system where decisions to add or remove nodes are made.

Figure 5.1 depicts a high level design of such autoscaler management node. All requests are received by the autoscaler and despatched to appropriate Cassandra nodes. The management node can use the techniques proposed in this thesis to measure the capacity of each replica and the degree of responsiveness based on the trade off between throughput and latency. Based on that new nodes can be added in order to balance the load and preserve the response time to be under a predefined value.

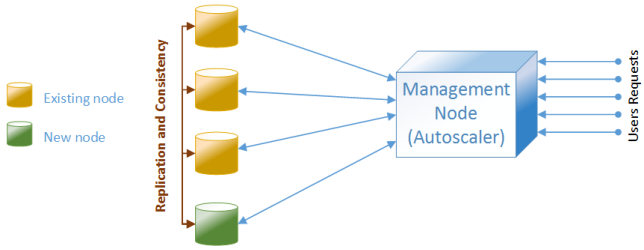



Figure 5.1: Cassandra nodes and Autoscaler management node

# Chapter 6

# Conclusion

In this thesis, I have presented a black-box technique that measures the performance of cloud applications. I probe the application remotely, iteratively adjusting the generated load based on the measured latency from previous steps. Using my technique, I have estimated the maximum capacity of an application for a given SLA over multiple cloud platforms. My results show that not only I can detect the performance differences between instance types and platforms, but I can also pinpoint individual VMs that unusually exhibit poor performance. Moreover, my methodology samples the server behaviour for a range of loads by recording the observed latencies for each load. From sampling results, I identify trade-off between performance and latency. I have also demonstrated the generic nature of my technique by testing it with a range of workloads and two different types of applications.

# Appendix A

# Cassandra Architecture for Read / Write

Cassandra is an open source distributed database management system. It is a massively scalable, decentralized data store. It has been designed to handle very large amounts of data spread out across many commodity servers while providing a highly available service with no single point of failure. Cassandra is a O(1)-hop routing Distributed Hash Table (DHT) which is eventually consistent but the consistency level can be tunable. Cassandra brings together the distributed systems technologies from Dynamo [DHJ+07] and the data model from Google's BigTable [CDG+08].

## A.1 Model Overview

Cassandra is a key-value data store, i.e., the data structure is a unique key and a collection of columns associated with a Column family. The primary units of information in Cassandra are described as follows:

- Column: A column is the atomic unit of information and is of the form name: value.

- Row: It is the uniquely identifiable data in the system which groups together columns. Every row in Cassandra is uniquely identifiable by its key.

- Column Family: A Column Family is the unit of abstraction containing keyed rows which group together columns of highly structured data. They have no defined schema of column names and types supported. It is equivalent to table in normal DBMS systems.

- Keyspace: The Keyspace is the top level unit of information in Cassandra. Column families are subordinate to exactly one keyspace. It is equivalent to database in normal DBMS systems.

## A.2 Architecture Elements

Cassandra uses both memory and disk to perform the basic operations, i.e., read and write. Before going deep into explaining how read and write paths look like, it is important to introduce memory and disk elements that are involved in read and write operations. Cassandra memory elements are explained as follows:

- Memtable: A sorted buffer that is created per column family and used at the time of write to store recent added data. It is also used in read operation and serve the recent stored data. By default, 1/3 of the heap will be considered to as memtable.

- Bloomfilter: A space-efficient probabilistic data structure that is used to test whether an element is a member of a set or not. False positives are possible, but false negatives are not. Bloom filter is used to quickly verify whether an existing disk (SSTable) contains a particular row without actually approaching the disk.

- Key Cache: A memory space that is used for read operation and hold the location of keys per-column family basis.

Cassandra disk elements are explained as follows:

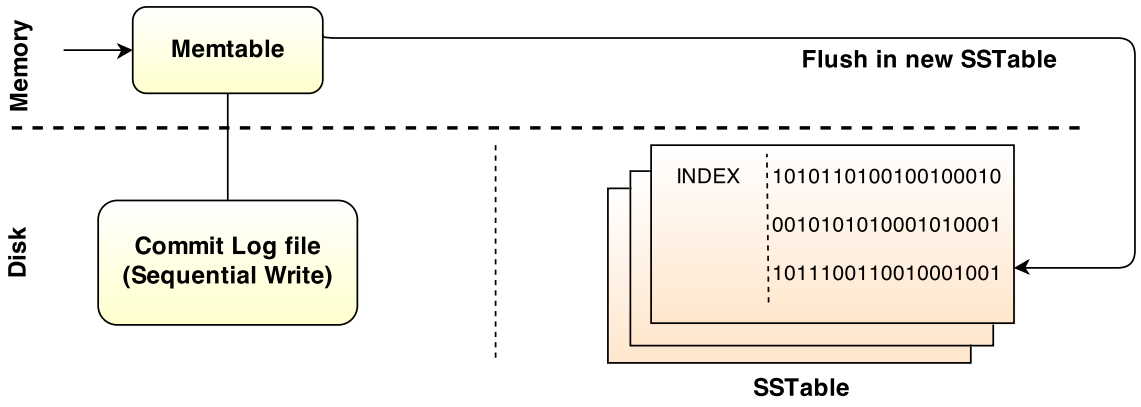



Figure A.1: Write path in Cassandra

- Commit log: This is a log file that is used by write operation. Data is written sequentially into commit log file. Commit log is important because in case of system crash, Cassandra would recreate memtables from commit log.

- SSTable (Sorted String Table): SSTable provides a persistent, ordered immutable map from keys to values, where both keys and values are arbitrary byte strings[CDG+08] and is maintained per table.

## A.3 Write Path

Write operation involves both memory and disk elements. To write a record Cassandra first store data in commit log and then memtable and reply back to client. Memtable can be seen as dedicated cache created individually for each column family. Memtable also contains all recent inserts. Each new insert for the same key and column will overwrite existing one. Multiple updates on single column will result in multiple entries in commit log, and single entry in memtable. Memtable will be flushed to disk (SSTable) when 1)it exceed maximum capacity 2)there are too many keys. Commit log is important as at the time of system crash, memtable would be created based on that. Flushing memtable creates immutable SSTable which simply save data to disk as sequential write. Compaction process will merge few SSTables into one to clean up, deleted data and merge together different modifications of single column. Before

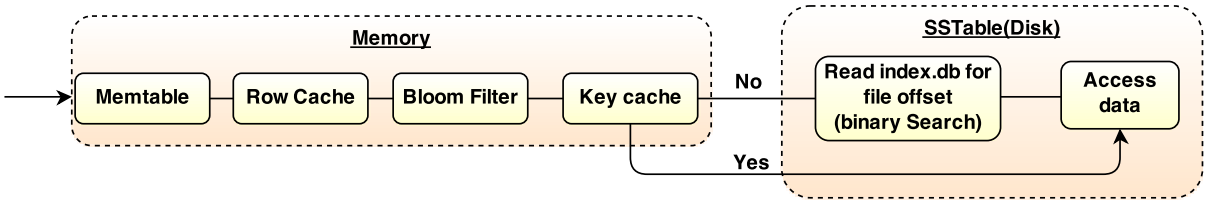



Figure A.2: Read path in Cassandra

compaction, a few SSTables could contain value of single column, after compaction it will be only one. Once the number of sstable reaches to a number (default is 4) compaction will happen. figure A.1 shows the write path and various disk and memory elements involved in this operation.

## A.4 Read Path

To read a record, Cassandra search a sequence of memory and disk locations and return a record. Figure A.2 depicts Cassandra read path. First, memtable is being searched, if requested record has been recently written, it is available from memtable and there is no need to search it from disk. This will improve the read performance.

If the data is not available in memtable, the row cache will be checked. If data is available it will be returned quickly from row cache. In other cases to quickly ensure data is in the disk, Cassandra use bloom filter. Bloom filters are used to save I/O operation when performing a key lookup: each SSTable has a bloom filter associated with it. This gives Cassandra the possibility to quickly verify, whether a given SSTable contains particular row. This will be checked before doing any disk seeks. Making queries for keys that don't exist in disk does not add any extra delay.

Now Cassandra have scanned all possible SSTables within particular column family, and found those with positive bloom filter for row key. Therefore it will sort them by last modification time and retrieve the data from most recent SSTable. Before approaching disk, one more check will happen. Key cache will be checked, a hit will lead directly from row key to column index.

If it does not exist, index.db file contains sorted row keys will be searched (binary search) for the column index. Finally the data will be retrieved from disk using the index.

# Appendix B

# Replication and Consistency in Geo-Distributed Data store

A number of systems that replicate data across geographically distributed data-centers (DCs) have emerged in recent years [LM10, DHJ+07, CDE+13, EWS12, CRS+08, LFKA13, BBC+11]. An important requirement on these systems is the need to support consistent updates on distributed replicas, and ensure both low read and write latencies. This is necessary as those systems work as back-ends for interactive web applications and serve reads and writes requests of geographically distributed end users (e.g., Facebook timelines, collaborative editing). Such geo-distributed applications are more complicated than simple replica servers which contain static content. Achieving low read and write latencies while meeting the consistency requirements is challenging. Replication and consistency might affect the overall perceived latency due to extra cross data-center latency. to maintain consistency across distributed replicas. In this section, I will provide relevant information about the replication and consistency in geo-distributed data-stores.

A commonly used scheme for geo-replicated data was to use a master-slave system, with master and slave replicas located in different DCs, and data asynchronously copied to the slave. However, slaves may not be completely synchronized with the master when a failure occurs. The system might serve stale data during the failure, and application-level reconciliation may be

required once the master recovers [Mor]. On the other hand, synchronized master-slave systems ensure consistency but face higher write latencies.

To address these limitations with master-slave systems, many geo-distributed cloud storage systems [CDE+13, BBC+11, LFKA13, BAC+13, SPAL11, KPF+13, LFKA11, EWS12] have been developed in the recent years. A distinguishing aspect of cloud datastores is the use of algorithms (e.g., quorum protocols [DHJ+07, LM10]) to maintain consistency across distributed replicas. Many geo-distributed datastores such as Dynamo [DHJ+07], and Cassandra [LM10] sacrifice stronger consistency for greater availability. Such systems offer tunable consistency at the level of operations. For example, consistency level of read (or write) can be set in [LM10] at different levels from weak level like *any* (if acknowledgement received from any replica, the operation is successful), *Two* (if two acknowledgements received from replicas, the operation is successful), *Three* to moderate level like *Quorum* and strong level like *all* (if all of replicas acknowledge, the operation is successful).

*Quorum-based datastores :* Quorum protocols have been extensively used in the distributed systems community for managing replicated data [Gif79]. Under quorum replication, the datastore writes a data item by sending it to a set of replicas (called a write quorum) and reads a data item by fetching it from a possibly different set of replicas (called a read quorum). While classical quorum protocols [Gif79] guarantee strong consistency, many geo-distributed datastores such as Dynamo [DHJ+07], and Cassandra [LM10] employ adapted versions of the quorum protocol, and sacrifice stronger consistency for greater availability [DHJ+07]. In these systems, reads (or writes) are sent to all replicas, and the read (or write) is treated successful if acknowledgements are received from a quorum. In case the replicas do not agree on the value of the item on a read, typically, the most recent value is returned to the user [DHJ+07, LM10], and a background process is used to propagate this value to other replicas.

The Quorum level writes to number of nodes that make up a quorum. A quorum is calculated, and then rounded down to a whole number, as follows:

$(Sum\_of\_Replication\_Factors\ /\ 2) + 1$

For example, in a three data centers cluster where each data center set a replication factor of 1, a quorum is 2. It means that read (or write) operation is successful if 2 replicas perform and return successfully.

The most common way of using the above scenario in such systems is when replication is configured in order to satisfy the *strict quorum property*. *Strict quorum property*, ensures that any read and write quorum of a data item intersect. Configuring replication with the strict quorum property in Cassandra and Dynamo guarantees read-your-writes consistency [Vog09]. Further, any read to a data item sees no version older than the last complete successful write for that item (though it may see any later write that is unsuccessful or is partially complete). Strict quorum property is defined as follows:

$R + W < N$

where N is the number of replicas, R and W are the read and write quorum sizes respectively. Furthermore, Dynamo and Cassandra can be explicitly configured with weaker quorum requirements leading to even weaker consistency guarantees [BVF+12].

# Bibliography


[Aka14] Akamai, http://www.akamai.com/, 2014.

[AL09] Sharad Agarwal and Jacob R Lorch. Matchmaking for online games and other latency-sensitive p2p systems. In *ACM SIGCOMM Computer Communication Review*, volume 39, pages 315–326. ACM, 2009.

[Ama14a] Amazon ec2 service level agreement. `http://aws.amazon.com/ec2-sla/.`, 2014.

[Ama14b] Amazon web services (aws). `http://aws.amazon.com/.`, 2014.

[Azu14] Microsoft azure. `http://azure.microsoft.com/en-us/.`, 2014.

[BAC+13] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry C Li, et al. Tao: Facebook's distributed data store for the social graph. In *USENIX Annual Technical Conference*, pages 49–60, 2013.

[BBC+11] Jason Baker, Chris Bond, James C Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Léon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *CIDR*, volume 11, pages 223–234, 2011.

[Bon01] Bonnie. http://www.textuality.com/bonnie/intro.html, 2001.

[BP98] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. *Computer networks and ISDN systems*, 30(1):107–117, 1998.

[Bri95] Thomas Brisco. Dns support for load balancing. 1995.

[BS10] Sean Kenneth Barker and Prashant Shenoy. Empirical evaluation of latency-sensitive application performance in the cloud. In *Proceedings of the first annual ACM SIGMM conference on Multimedia systems*, pages 35–46. ACM, 2010.

[BVF+12] Peter Bailis, Shivaram Venkataraman, Michael J Franklin, Joseph M Hellerstein, and Ion Stoica. Probabilistically bounded staleness for practical partial quorums. *Proceedings of the VLDB Endowment*, 5(8):776–787, 2012.

[cas14] Cassjmeter. cassandra jmeter plugin. `https://github.com/Netflix/CassJMeter`, 2014.

[CCR+03] Brent Chun, David Culler, Timothy Roscoe, Andy Bavier, Larry Peterson, Mike Wawrzoniak, and Mic Bowman. Planetlab: an overlay testbed for broad-coverage services. *ACM SIGCOMM Computer Communication Review*, 33(3):3–12, 2003.

[CCY00] Valeria Cardellini, Michele Colajanni, and Philip S Yu. Geographic load balancing for scalable distributed web systems. In *Modeling, Analysis and Simulation of Computer and Telecommunication Systems, 2000. Proceedings. 8th International Symposium on*, pages 20–27. IEEE, 2000.

[CDE+13] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. Spanner: Googles globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):8, 2013.

[CDG+08] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[CGGK11] Yanpei Chen, Archana Ganapathi, Rean Griffith, and Randy Katz. The case for evaluating mapreduce performance using workload suites. In *Modeling, Analysis*

*& Simulation of Computer and Telecommunication Systems (MASCOTS), 2011 IEEE 19th International Symposium on*, pages 390–399. IEEE, 2011.

[CGP00] Marco Conti, Enrico Gregori, and Fabio Panzieri. Load distribution among replicated web servers: A qos-based approach. *ACM SIGMETRICS Performance Evaluation Review*, 27(4):12–19, 2000.

[CKK02] Yan Chen, Randy H Katz, and John D Kubiatowicz. Dynamic replica placement for scalable content delivery. In *Peer-to-peer systems*, pages 306–318. Springer, 2002.

[CP09] Alistair Croll and Sean Power. *Complete Web Monitoring: Watching your visitors, performance, communities, and competitors.* " O'Reilly Media, Inc.", 2009.

[CRS+08] Brian F Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *Proceedings of the VLDB Endowment*, 1(2):1277–1288, 2008.

[CST+10] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154. ACM, 2010.

[CYC98] Michele Colajanni, PS Yu, and Valeria Cardellini. Dynamic load balancing in geographically distributed heterogeneous web servers. In *Distributed Computing Systems, 1998. Proceedings. 18th International Conference on*, pages 295–302. IEEE, 1998.

[CYD97] Michele Colajanni, Philip S. Yu, and Daniel M Dias. Scheduling algorithms for distributed web servers. In *Distributed Computing Systems, 1997., Proceedings of the 17th International Conference on*, pages 169–176. IEEE, 1997.

[dbe] Dbench. `https://www.samba.org/ftp/tridge/dbench/`.

[DCKM04] Frank Dabek, Russ Cox, Frans Kaashoek, and Robert Morris. Vivaldi: A decentralized network coordinate system. In *ACM SIGCOMM Computer Communication Review*, volume 34, pages 15–26. ACM, 2004.

[DHJ+07] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 205–220. ACM, 2007.

[Dix09] P. Dixon. Shopzilla's site redo - you get what you measure, velocity conference talk. `http://velocityconf.com/velocity2009/public/schedule/detail/7709`, 2009.

[DMP+02] John Dilley, Bruce Maggs, Jay Parikh, Harald Prokop, Ramesh Sitaraman, and Bill Weihl. Globally distributed content delivery. *Internet Computing, IEEE*, 6(5):50–58, 2002.

[Dyn14] Dyndns, a cloud-based dns hosting platforms. http://dyn.com/dns/, 2014.

[EWS12] Robert Escriva, Bernard Wong, and Emin Gün Sirer. Hyperdex: A distributed, searchable key-value store. *ACM SIGCOMM Computer Communication Review*, 42(4):25–36, 2012.

[FAK+12] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. Clearing the clouds: a study of emerging scale-out workloads on modern hardware. *ACM SIGARCH Computer Architecture News*, 40(1):37–48, 2012.

[FBZA] Zongming Fei, Samrat Bhattacharjee, Ellen W Zegura, and Mostafa H Ammar. A novel server selection technique for improving the response time of a replicated service. In *INFOCOM'98. Seventeenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 2, pages 783–791. IEEE.

[FJJ+01] Paul Francis, Sugih Jamin, Cheng Jin, Yixin Jin, Danny Raz, Yuval Shavitt, and Lixia Zhang. Idmaps: A global internet host distance estimation service. *Networking, IEEE/ACM Transactions on*, 9(5):525–540, 2001.

[FJV+12] Benjamin Farley, Ari Juels, Venkatanathan Varadarajan, Thomas Ristenpart, Kevin D Bowers, and Michael M Swift. More for your money: exploiting performance heterogeneity in public clouds. In *Proceedings of the Third ACM Symposium on Cloud Computing*, page 20. ACM, 2012.

[FRE06] MJ FREEDMAN. Oasis: Anycast for any service. In *Proc. of Symp. on NSDI, USENIX, 2006*, 2006.

[GAE14] Google appengine. `https://appengine.google.com/.`, 2014.

[gce14] Google compute engine. `https://cloud.google.com/products/compute-engine/.`, 2014.

[Geo04] geobkend, http://doc.powerdns.com/html/geo.html/, 2004.

[Gif79] David K Gifford. Weighted voting for replicated data. In *Proceedings of the seventh ACM symposium on Operating systems principles*, pages 150–162. ACM, 1979.

[gri] Awslb, http://hadoop.apache.org/mapreduce/docs/current/gridmix.html/.

[GS95] James D Guyton and Michael F Schwartz. *Locating nearby copies of replicated Internet servers*, volume 25. ACM, 1995.

[GSG02] Krishna P Gummadi, Stefan Saroiu, and Steven D Gribble. King: Estimating latency between arbitrary internet end hosts. In *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment*, pages 5–18. ACM, 2002.

[Ham09] James Hamilton. The cost of latency. *Perspectives Blog*, 2009.

[HHD+10] Shengsheng Huang, Jie Huang, Jinquan Dai, Tao Xie, and Bo Huang. The hibench benchmark suite: Characterization of the mapreduce-based data analysis. In *Data*

*Engineering Workshops (ICDEW), 2010 IEEE 26th International Conference on*, pages 41–51. IEEE, 2010.

[IOY+11] Alexandru Iosup, Simon Ostermann, M Nezih Yigitbasi, Radu Prodan, Thomas Fahringer, and Dick HJ Epema. Performance analysis of cloud computing services for many-tasks scientific computing. *Parallel and Distributed Systems, IEEE Transactions on*, 22(6):931–945, 2011.

[jme14a] Apache jmeter. `https://jmeter.apache.org/.`, 2014.

[jme14b] Jmeter plugins. `http://jmeter-plugins.org/.`, 2014.

[JRM+10] Keith R Jackson, Lavanya Ramakrishnan, Krishna Muriki, Shane Canon, Shreyas Cholia, John Shalf, Harvey J Wasserman, and Nicholas J Wright. Performance analysis of high performance computing applications on the amazon web services cloud. In *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, pages 159–168. IEEE, 2010.

[jso14] Json-rpc, http://json-rpc.org/, 2014.

[KLL+97] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 654–663. ACM, 1997.

[KM02] Magnus Karlsson and Mallik Mahalingam. Do we need replica placement algorithms in content delivery networks. 2002.

[KMS+09] Rupa Krishnan, Harsha V Madhyastha, Sridhar Srinivasan, Sushant Jain, Arvind Krishnamurthy, Thomas Anderson, and Jie Gao. Moving beyond end-to-end path information to optimize cdn performance. In *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference*, pages 190–201. ACM, 2009.

[KPF+13] Tim Kraska, Gene Pang, Michael J Franklin, Samuel Madden, and Alan Fekete. Mdcc: Multi-data center consistency. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 113–126. ACM, 2013.

[LFKA11] Wyatt Lloyd, Michael J Freedman, Michael Kaminsky, and David G Andersen. Don't settle for eventual: scalable causal consistency for wide-area storage with cops. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 401–416. ACM, 2011.

[LFKA13] Wyatt Lloyd, Michael J Freedman, Michael Kaminsky, and David G Andersen. Stronger semantics for low-latency geo-replicated storage. In *NSDI*, pages 313–328, 2013.

[Lin06] Greg Linden. Make data useful. *Presentation, Amazon, November*, 2006.

[LM10] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[LYKZ10] Ang Li, Xiaowei Yang, Srikanth Kandula, and Ming Zhang. Cloudcmp: comparing public cloud providers. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, pages 1–14. ACM, 2010.

[LZJ+12] Chunjie Luo, Jianfeng Zhan, Zhen Jia, Lei Wang, Gang Lu, Lixin Zhang, Cheng-Zhong Xu, and Ninghui Sun. Cloudrank-d: benchmarking and ranking cloud computing systems for data processing applications. *Frontiers of Computer Science*, 6(4):347–362, 2012.

[MIP+06] Harsha V Madhyastha, Tomas Isdal, Michael Piatek, Colin Dixon, Thomas Anderson, Arvind Krishnamurthy, and Arun Venkataramani. iplane: An information plane for distributed services. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 367–380. USENIX Association, 2006.

[Mor] More 9s please: Under the covers of the high replication datastore, http://www.google.com/events/io/2011/sessions/more-9s-please-under-the-covers-of-the-high-replication-datastore.html/.

[Nag14] Nagios. http://www.nagios.org/, 2014.

[Neu14] Neustar,geopoint - ip geolocation experts, http://www.neustar.biz/resources/tools/ip-geolocation-lookup-tool/., 2014.

[NOZ+06] Ramanathan Narayanan, Berkin Ozisikyilmaz, Joseph Zambreno, Gokhan Memik, and Alok Choudhary. Minebench: A benchmark suite for data mining workloads. In *Workload Characterization, 2006 IEEE International Symposium on*, pages 182–188. IEEE, 2006.

[NZ02] TS Eugene Ng and Hui Zhang. Predicting internet network distance with coordinates-based approaches. In *INFOCOM 2002. Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 1, pages 170–179. IEEE, 2002.

[OIY+10] Simon Ostermann, Alexandria Iosup, Nezih Yigitbasi, Radu Prodan, Thomas Fahringer, and Dick Epema. A performance analysis of ec2 cloud computing services for scientific computing. In *Cloud Computing*, pages 115–131. Springer, 2010.

[OZN+] Zhonghong Ou, Hao Zhuang, Jukka K Nurminen, Antti Ylä-Jääski, and Pan Hui. Exploiting hardware heterogeneity within the same instance type of amazon ec2.

[PAS+04] Jeffrey Pang, Aditya Akella, Anees Shaikh, Balachander Krishnamurthy, and Srinivasan Seshan. On the responsiveness of dns-based network control. In *Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*, pages 21–26. ACM, 2004.

[PS01] Venkata N Padmanabhan and Lakshminarayanan Subramanian. An investigation of geographic mapping techniques for internet hosts. In *ACM SIGCOMM Computer Communication Review*, volume 31, pages 173–185. ACM, 2001.

[PZMH07] Himabindu Pucha, Ying Zhang, Z. Morley Mao, and Y. Charlie Hu. Understanding network delay changes caused by routing events. In *Prof. of ACM SIGMETRICS*, pages 73–84, 2007.

[QPV01] Lili Qiu, Venkata N Padmanabhan, and Geoffrey M Voelker. On the placement of web server replicas. In *INFOCOM 2001. Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 3, pages 1587–1596. IEEE, 2001.

[Rac14] Rackspace cloud. `http://www.rackspace.co.uk/`, 2014.

[RKK04] Supranamaya Ranjan, Roger Karrer, and E Knightly. Wide area redirection of dynamic content by internet data centers. In *INFOCOM 2004. Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies*, volume 2, pages 816–826. IEEE, 2004.

[Rou14] Amazon route53, http://aws.amazon.com/route53/, 2014.

[SB09] Eric Schurman and Jake Brutlag. The user and business impact of server delays, additional bytes, and http chunking in web search. In *Velocity Web Performance and Operations Conference*, 2009.

[SCKB06] Ao-Jan Su, David R Choffnes, Aleksandar Kuzmanovic, and Fabián E Bustamante. Drafting behind akamai (travelocity-based detouring). *ACM SIGCOMM Computer Communication Review*, 36(4):435–446, 2006.

[SDQR10] Jörg Schad, Jens Dittrich, and Jorge-Arnulfo Quiané-Ruiz. Runtime measurements in the cloud: observing, analyzing, and reducing variance. *Proceedings of the VLDB Endowment*, 3(1-2):460–471, 2010.

[SPAL11] Yair Sovran, Russell Power, Marcos K Aguilera, and Jinyang Li. Transactional storage for geo-replicated systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 385–400. ACM, 2011.

[spe] Spec java virtual machine benchmark 2008., http://www.spec.org/jvm2008/.

[SSS+08] Will Sobel, Shanti Subramanyam, Akara Sucharitakul, Jimmy Nguyen, Hubert Wong, Arthur Klepchukov, Sheetal Patil, Armando Fox, and David Patterson.

Cloudstone: Multi-platform, multi-language benchmark and measurement tools for web 2.0. In *Proc. of CCA*, volume 8, 2008.

[STA01] Anees Shaikh, Renu Tewari, and Mukesh Agrawal. On the effectiveness of dns-based server selection. In *INFOCOM 2001. Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 3, pages 1801–1810. IEEE, 2001.

[tpc14] Tpc-c, http://www.tpc.org/tpcc/, 2014.

[tpi] tcpping. `http://www.vdberg.org/~richard/tcpping.html/`.

[Ube01] Ubench. http://phystech.com/download/ubench.html., 2001.

[uni] Unixbench. `http://freecode.com/projects/unixbench/`.

[UPvS09] Guido Urdaneta, Guillaume Pierre, and Maarten van Steen. Wikipedia workload analysis for decentralized hosting. *Elsevier Computer Networks*, 53(11):1830–1845, July 2009. `http://www.globule.org/publi/WWADH_comnet2009.html`.

[Vog09] Werner Vogels. Eventually consistent. *Communications of the ACM*, 52(1):40–44, 2009.

[VRMCL08] Luis M Vaquero, Luis Rodero-Merino, Juan Caceres, and Maik Lindner. A break in the clouds: towards a cloud definition. *ACM SIGCOMM Computer Communication Review*, 39(1):50–55, 2008.

[WAz14] Windows azure traffic manager (watm), http://azure.microsoft.com/en-us/services/traffic-manager/, 2014.

[Wik05] The technology behind powerdns hosting. https://www.powerdns.net/en/content/dns-technology.aspx, 2005.

[WJFR10] Patrick Wendell, Joe Wenjie Jiang, Michael J Freedman, and Jennifer Rexford. Donar: decentralized server selection for cloud services. *ACM SIGCOMM Computer Communication Review*, 40(4):231–242, 2010.

[WN10] Guohui Wang and TS Eugene Ng. The impact of virtualization on network performance of amazon ec2 data center. In *INFOCOM, 2010 Proceedings IEEE*, pages 1–9. IEEE, 2010.

[WPP02] Limin Wang, Vivek Pai, and Larry Peterson. The effectiveness of request redirection on cdn robustness. *ACM SIGOPS Operating Systems Review*, 36(SI):345–360, 2002.

[WSS05] Bernard Wong, Aleksandrs Slivkins, and Emin Gün Sirer. Meridian: A lightweight network location service without virtual coordinates. *ACM SIGCOMM Computer Communication Review*, 35(4):85–96, 2005.

[Zab14] Zabbix. http://www.zabbix.com/, 2014.